PACKT PUBLISHING

用Python写网络爬虫

Web Scraping with Python

Scrape data from any website with the power of Python

[澳] Richard Lawson 著

李斌 译

中国工信出版集团

人民邮电出版社 POSTS & TELECOM PRESS

用Python写网络爬虫

[澳] Richard Lawson 著
李斌 译

人民邮电出版社
北京

图书在版编目（CIP）数据

用Python写网络爬虫 /（澳大利亚）理查德·劳森（Richard Lawson）著；李斌译. -- 北京：人民邮电出版社，2016.9（2016.11重印）
ISBN 978-7-115-43179-0

Ⅰ. ①用… Ⅱ. ①理… ②李… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2016)第177976号

版权声明

◆ 著　[澳] Richard Lawson
译　李　斌
责任编辑　傅道坤
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：800×1000　1/16
印张：10.75
字数：148 千字　2016 年 9 月第 1 版
印数：7 001－11 000 册　2016 年 11 月河北第 4 次印刷

著作权合同登记号　图字：01-2016-3962 号

定价：45.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

内容提要

本书讲解了如何使用 Python 来编写网络爬虫程序，内容包括网络爬虫简介，从页面中抓取数据的三种方法，提取缓存中的数据，使用多个线程和进程来进行并发抓取，如何抓取动态页面中的内容，与表单进行交互，处理页面中的验证码问题，以及使用 Scarpy 和 Portia 来进行数据抓取，并在最后使用本书介绍的数据抓取技术对几个真实的网站进行了抓取，旨在帮助读者活学活用书中介绍的技术。

本书适合有一定 Python 编程经验，而且对爬虫技术感兴趣的读者阅读。

关于作者

Richard Lawson 来自澳大利亚，毕业于墨尔本大学计算机科学专业。毕业后，他创办了一家专注于网络爬虫的公司，为超过 50 个国家的业务提供远程工作。他精通于世界语，可以使用汉语和韩语对话，并且积极投身于开源软件。他目前在牛津大学攻读研究生学位，并利用业余时间研发自主无人机。

我要感谢 Timothy Baldwin 教授将我引入这个令人兴奋的领域，以及本书编写时在巴黎招待我的 Tharavy Douc。

关于审稿人

Martin Burch 是一名常驻纽约的数据记者，其工作是为华尔街日报绘制交互式图表。他在新墨西哥州立大学获得了新闻学和信息系统专业的学士学位，然后在纽约城市大学新闻学研究院获得了新闻学专业硕士学位。

我要感谢我的妻子 Lisa 鼓励我协助本书的创作，我的叔叔 Michael 耐心解答我的编程问题，以及我的父亲 Richard 激发了我对新闻学和写作的热爱。

William Sankey 是一位数据专业人士，也是一位业余开发人员，生活在马里兰州科利奇帕克市。他于 2012 年毕业于约翰·霍普金斯大学，获得了公共政策硕士学位，专业方向为定量分析。他目前在 L&M 政策研究有限责任公司担任健康服务研究员，从事与美国医疗保险和医疗补助服务中心（CMS）相关的项目。这些项目包括责任医疗机构评估以及精神病院住院患者预付费系统监测。

我要感谢我深爱的妻子 Julia 和顽皮的小猫 Ruby，给予我全部的爱和支持。

Ayush Tiwari 是一名 Python 开发者，本科就读于印度理工学院罗克分校。他自 2013 年起工作于印度理工学院罗克分校信息管理小组，并活跃于网络开发领域。对他而言，审阅本书是一个非常棒的经历。他不仅是一名审稿人，也是一名狂热的网络爬虫学习者。他向所有 Python 爱好者推荐本书，以便享受爬虫的益处。

他热衷于 Python 网络爬虫，曾参与体育直播订阅、通用 Python 电子商务网络爬虫（在 Miranj）等相关项目。

他还使用 Django 应用开发了就业门户，帮助改善印度理工学院罗克分校的就业流程。

除了后端开发之外，他还喜欢使用诸如 NumPy、SciPy 等 Python 库进行科学计算和数据分析，目前他从事计算流体力学领域的研究。你可以在 GitHub 上访问到他的项目，他的用户名是 `tiwariayush`。

他喜欢徒步穿越喜马拉雅山谷，每年会参加多次徒步行走活动。此外，他还喜欢弹吉他。他的成就还包括参加国际知名的 Super 30 小组，并在其中成为排名保持者。他在高中时，还参加了国际奥林匹克数学竞赛。

我的家庭成员（我的姐姐 Aditi、我的父母以及 Anand 先生）、我在 VI 和 IMG 的朋友以及我的教授都为我提供了很大的帮助。我要感谢他们所有人对我的支持。

最后，感谢尊敬的作者和 Packt 出版社团队出版了这些非常好的技术书籍。我要对他们在编写这些书籍时的所有辛苦工作表示赞赏。

前言

互联网包含了迄今为止最有用的数据集，并且大部分可以免费公开访问。但是，这些数据难以复用。它们被嵌入在网站的结构和样式当中，需要抽取出来才能使用。从网页中抽取数据的过程又被称为网络爬虫。随着越来越多的信息被发布到网络上，网络爬虫也变得越来越有用。

本书内容

第 1 章，网络爬虫简介，介绍了网络爬虫，并讲解了爬取网站的方法。

第 2 章，数据抓取，展示了如何从网页中抽取数据。

第 3 章，下载缓存，学习了如何通过缓存结果避免重复下载的问题。

第 4 章，并发下载，通过并行下载加速数据抓取。

第 5 章，动态内容，展示了如何从动态网站中抽取数据。

第 6 章，表单交互，展示了如何与表单进行交互，从而访问你需要的数据。

第 7 章，验证码处理，阐述了如何访问被验证码图像保护的数据。

第 8 章，Scrapy，学习了如何使用流行的高级框架 Scrapy。

第 9 章，总结，对我们介绍的这些网络爬虫技术进行总结。

阅读本书的前提

本书中所有的代码都已经在 Python 2.7 环境中进行过测试，并且可以从 http://bitbucket.org/wswp/code 下载到这些源代码。理想情况下，本书未来的版本会将示例代码移植到 Python 3 当中。不过，现在依赖的很多库（比如 Scrapy/Twisted、Mechanize 和 Ghost）还只支持 Python 2。为了帮助阐明爬取示例，我们创建了一个示例网站，其网址为 http://example.webscraping.com。由于该网站限制了下载内容的速度，因此如果你希望自行搭建示例网站，可以从 http://bitbucket.org/wswp/places 获取网站源代码和安装说明。

我们决定为本书中使用的大部分示例搭建一个定制网站，而不是抓取活跃网站，这样我们就对环境拥有了完全控制。这种方式为我们提供了稳定性，因为活跃网站要比书中的定制网站更新更加频繁，并且当你尝试运行爬虫示例时，代码可能已经无法工作。另外，定制网站允许我们自定义示例，用于阐释特定技巧并避免其他干扰。最后，活跃网站可能并不欢迎我们使用它作为学习网络爬虫的对象，并且可能会尝试封禁我们的爬虫。使用我们自己定制的网站可以规避这些风险，不过在这些例子中学到的技巧确实也可以应用到这些活跃网站当中。

本书读者

阅读本书需要有一定的编程经验，并且不适用于绝对的初学者。在实践中，我们将会首先实现我们自己的网络爬虫技术版本，然后才会介绍现有的流行模块，这样可以让你更好地理解这些技术是如何工作的。本书中的这些示例将假设你已经拥有 Python 语言以及使用 pip 安装模块的能力。如果你想复习一下这些知识，有一本非常好的免费在线书籍可以使用，其作者为 Mark

Pilgrim，书籍网址是 http://www.diveintopython.net。这本书也是我初学 Python 时所使用的资源。

此外，这些例子还假设你已经了解网页是如何使用 HTML 进行构建并通过 JavaScript 更新的知识。关于 HTTP、CSS、AJAX、WebKit 以及 MongoDB 的既有知识也很有用，不过它们不是必需的，这些技术会在需要使用时进行介绍。上述很多主题的详细参考资料可以从 http://www.w3schools.com 获取到。

目录

第 1 章
网络爬虫简介

本章中，我们将会介绍如下主题：

- 网络爬虫领域简介；
- 解释合法性质疑；
- 对目标网站进行背景调研；
- 逐步完善一个高级网络爬虫。

1.1 网络爬虫何时有用

假设我有一个鞋店，并且想要及时了解竞争对手的价格。我可以每天访问他们的网站，与我店铺中鞋子的价格进行对比。但是，如果我店铺中的鞋类品种繁多，或是希望能够更加频繁地查看价格变化的话，就需要花费大量的时间，甚至难以实现。再举一个例子，我看中了一双鞋，想等它促销时再购买。我可能需要每天访问这家鞋店的网站来查看这双鞋是否降价，也许需要等待几个月的时间，我才能如愿盼到这双鞋促销。上述这两个重复性的手工流程，都可以利用本书介绍的网络爬虫技术实现自动化处理。

理想状态下，网络爬虫并不是必须品，每个网站都应该提供 API，以结构化的格式共享它们的数据。然而现实情况中，虽然一些网站已经提供了这种 API，但是它们通常会限制可以抓取的数据，以及访问这些数据的频率。另外，对于网站的开发者而言，维护前端界面比维护后端 API 接口优先级更高。总之，我们不能仅仅依赖于 API 去访问我们所需的在线数据，而是应该学习一些网络爬虫技术的相关知识。

1.2　网络爬虫是否合法

网络爬虫目前还处于早期的蛮荒阶段，“允许哪些行为”这种基本秩序还处于建设之中。从目前的实践来看，如果抓取数据的行为用于个人使用，则不存在问题；而如果数据用于转载，那么抓取的数据类型就非常关键了。

世界各地法院的一些案件可以帮助我们确定哪些网络爬虫行为是允许的。在 *Feist Publications, Inc.*起诉 *Rural Telephone Service Co.*的案件中，美国联邦最高法院裁定抓取并转载真实数据（比如，电话清单）是允许的。而在澳大利亚，*Telstra Corporation Limited* 起诉 *Phone Directories Company Pty Ltd* 这一类似案件中，则裁定只有拥有明确作者的数据，才可以获得版权。此外，在欧盟的 *ofir.dk* 起诉 *home.dk* 一案中，最终裁定定期抓取和深度链接是允许的。

这些案件告诉我们，当抓取的数据是现实生活中的真实数据（比如，营业地址、电话清单）时，是允许转载的。但是，如果是原创数据（比如，意见和评论），通常就会受到版权限制，而不能转载。

无论如何，当你抓取某个网站的数据时，请记住自己是该网站的访客，应当约束自己的抓取行为，否则他们可能会封禁你的 IP，甚至采取更进一步的法律行动。这就要求下载请求的速度需要限定在一个合理值之内，并且还需要设定一个专属的用户代理来标识自己。在下面的小节中我们将会对这些实践进行具体介绍。

关于上述几个法律案件的更多信息可以参考下述地址：

- http://caselaw.lp.findlaw.com/scripts/getcase. pl?court=US&vol=499&invol=340
- http://www.austlii.edu.au/au/cases/cth/FCA/2010/44.html
- http://www.bvhd.dk/uploads/tx_mocarticles/S_og_Handelsrettens_afg_relse_i_Ofir-sagen.pdf

1.3 背景调研

在深入讨论爬取一个网站之前，我们首先需要对目标站点的规模和结构进行一定程度的了解。网站自身的 robots.txt 和 Sitemap 文件都可以为我们提供一定的帮助，此外还有一些能提供更详细信息的外部工具，比如 Google 搜索和 WHOIS。

1.3.1 检查 robots.txt

大多数网站都会定义 robots.txt 文件，这样可以让爬虫了解爬取该网站时存在哪些限制。这些限制虽然仅仅作为建议给出，但是良好的网络公民都应当遵守这些限制。在爬取之前，检查 robots.txt 文件这一宝贵资源可以最小化爬虫被封禁的可能，而且还能发现和网站结构相关的线索。关于 robots.txt 协议的更多信息可以参见 http://www.robotstxt.org。下面的代码是我们的示例文件 robots.txt 中的内容，可以访问 http://example.webscraping.com/robots.txt 获取。

```
# section 1
User-agent: BadCrawler
Disallow: /
```

```
# section 2
User-agent: *
Crawl-delay: 5
Disallow: /trap

# section 3
Sitemap: http://example.webscraping.com/sitemap.xml
```

在 section 1 中，robots.txt 文件禁止用户代理为 BadCrawler 的爬虫爬取该网站，不过这种写法可能无法起到应有的作用，因为恶意爬虫根本不会遵从 robots.txt 的要求。本章后面的一个例子将会展示如何让爬虫自动遵守 robots.txt 的要求。

section 2 规定，无论使用哪种用户代理，都应该在两次下载请求之间给出 5 秒的抓取延迟，我们需要遵从该建议以避免服务器过载。这里还有一个/trap 链接，用于封禁那些爬取了不允许链接的恶意爬虫。如果你访问了这个链接，服务器就会封禁你的 IP 一分钟！一个真实的网站可能会对你的 IP 封禁更长时间，甚至是永久封禁。不过如果这样设置的话，我们就无法继续这个例子了。

section 3 定义了一个 Sitemap 文件，我们将在下一节中了解如何检查该文件。

1.3.2　检查网站地图

网站提供的 Sitemap 文件（即网站地图）可以帮助爬虫定位网站最新的内容，而无须爬取每一个网页。如果想要了解更多信息，可以从 http://www.sitemaps.org/protocol.html 获取网站地图标准的定义。下面是在 robots.txt 文件中发现的 Sitemap 文件的内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url><loc>http://example.webscraping.com/view/Afghanistan-1
    </loc></url>
  <url><loc>http://example.webscraping.com/view/Aland-Islands-2
```

```
    </loc></url>
    <url><loc>http://example.webscraping.com/view/Albania-3</loc>
    </url>
    ...
</urlset>
```

网站地图提供了所有网页的链接，我们会在后面的小节中使用这些信息，用于创建我们的第一个爬虫。虽然 Sitemap 文件提供了一种爬取网站的有效方式，但是我们仍需对其谨慎处理，因为该文件经常存在缺失、过期或不完整的问题。

1.3.3 估算网站大小

目标网站的大小会影响我们如何进行爬取。如果是像我们的示例站点这样只有几百个 URL 的网站，效率并没有那么重要；但如果是拥有数百万个网页的站点，使用串行下载可能需要持续数月才能完成，这时就需要使用第 4 章中介绍的分布式下载来解决了。

估算网站大小的一个简便方法是检查 Google 爬虫的结果，因为 Google 很可能已经爬取过我们感兴趣的网站。我们可以通过 Google 搜索的 site 关键词过滤域名结果，从而获取该信息。我们可以从 http://www.google.com/advanced_search 了解到该接口及其他高级搜索参数的用法。

图 1.1 所示为使用 site 关键词对我们的示例网站进行搜索的结果，即在 Google 中搜索 site:example.webscraping.com。

从图 1.1 中可以看出，此时 Google 估算该网站拥有 202 个网页，这和实际情况差不多。不过对于更大型的网站，我们会发现 Google 的估算并不十分准确。

在域名后面添加 URL 路径，可以对结果进行过滤，仅显示网站的某些部分。图 1.2 所示为搜索 site:example.webscraping.com/view 的结果。该搜索条件会限制 Google 只搜索国家页面。

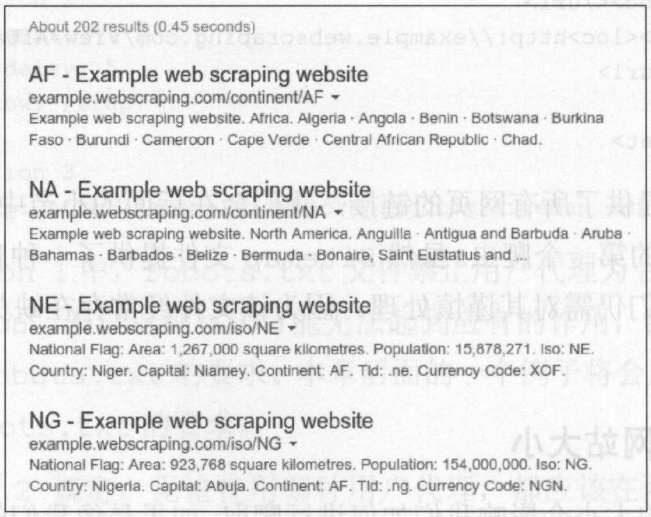


图 1.1

图 1.2

这种附加的过滤条件非常有用，因为在理想情况下，你只希望爬取网站中包含有用数据的部分，而不是爬取网站的每个页面。

1.3.4 识别网站所用技术

构建网站所使用的技术类型也会对我们如何爬取产生影响。有一个十分有用的工具可以检查网站构建的技术类型——builtwith 模块。该模块的安装方法如下。

```
pip install builtwith
```

该模块将 URL 作为参数，下载该 URL 并对其进行分析，然后返回该网站使用的技术。下面是使用该模块的一个例子。

```
>>> import builtwith
>>> builtwith.parse('http://example.webscraping.com')
{u'javascript-frameworks': [u'jQuery', u'Modernizr', u'jQuery UI'],
 u'programming-languages': [u'Python'],
 u'web-frameworks': [u'Web2py', u'Twitter Bootstrap'],
 u'web-servers': [u'Nginx']}
```

从上面的返回结果中可以看出，示例网站使用了 Python 的 Web2py 框架，另外还使用了一些通用的 JavaScript 库，因此该网站的内容很有可能是嵌入在 HTML 中的，相对而言比较容易抓取。而如果改用 AngularJS 构建该网站，此时的网站内容就很可能是动态加载的。另外，如果网站使用了 ASP.NET，那么在爬取网页时，就必须要用到会话管理和表单提交了。对于这些更加复杂的情况，我们会在第 5 章和第 6 章中进行介绍。

1.3.5 寻找网站所有者

对于一些网站，我们可能会关心其所有者是谁。比如，我们已知网站的所有者会封禁网络爬虫，那么我们最好把下载速度控制得更加保守一些。为了找到网站的所有者，我们可以使用 WHOIS 协议查询域名的注册者是谁。Python 中有一个针对该协议的封装库，其文档地址为 https://pypi.python.org/pypi/python-whois，我们可以通过 pip 进行安装。

```
pip install python-whois
```

下面是使用该模块对 appspot.com 这个域名进行 WHOIS 查询时的返回结果。

```
>>> import whois
>>> print whois.whois('appspot.com')
{
  ...
  "name_servers": [
    "NS1.GOOGLE.COM",
    "NS2.GOOGLE.COM",
    "NS3.GOOGLE.COM",
    "NS4.GOOGLE.COM",
    "ns4.google.com",
    "ns2.google.com",
    "ns1.google.com",
    "ns3.google.com"
  ],
"org": "Google Inc.",
"emails": [
    "abusecomplaints@markmonitor.com",
    "dns-admin@google.com"
  ]
}
```

从结果中可以看出该域名归属于 Google，实际上也确实如此。该域名是用于 Google App Engine 服务的。当我们爬取该域名时就需要十分小心，因为 Google 经常会阻断网络爬虫，尽管实际上其自身就是一个网络爬虫业务。

1.4　编写第一个网络爬虫

为了抓取网站，我们首先需要下载包含有感兴趣数据的网页，该过程一般被称为**爬取（crawling）**。爬取一个网站有很多种方法，而选用哪种方法更加合适，则取决于目标网站的结构。本章中，首先会探讨如何安全地下载网页，然后会介绍如下 3 种爬取网站的常见方法：

- 爬取网站地图；
- 遍历每个网页的数据库 ID；
- 跟踪网页链接。

1.4.1 下载网页

要想爬取网页，我们首先需要将其下载下来。下面的示例脚本使用 Python 的 urllib2 模块下载 URL。

```
import urllib2
def download(url):
    return urllib2.urlopen(url).read()
```

当传入 URL 参数时，该函数将会下载网页并返回其 HTML。不过，这个代码片段存在一个问题，即当下载网页时，我们可能会遇到一些无法控制的错误，比如请求的页面可能不存在。此时，urllib2 会抛出异常，然后退出脚本。安全起见，下面再给出一个更健壮的版本，可以捕获这些异常。

```
import urllib2

def download(url):
    print 'Downloading:', url
    try:
        html = urllib2.urlopen(url).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
        html = None
    return html
```

现在，当出现下载错误时，该函数能够捕获到异常，然后返回 None。

1. 重试下载

下载时遇到的错误经常是临时性的，比如服务器过载时返回的 503 Service Unavailable 错误。对于此类错误，我们可以尝试重新下载，因为这个服务器问题现在可能已解决。不过，我们不需要对所有错误都尝试重

新下载。如果服务器返回的是 404 Not Found 这种错误，则说明该网页目前并不存在，再次尝试同样的请求一般也不会出现不同的结果。

互联网工程任务组（Internet Engineering Task Force）定义了 HTTP 错误的完整列表，详情可参考 https://tools.ietf.org/html/rfc7231#section-6。从该文档中，我们可以了解到 4xx 错误发生在请求存在问题时，而 5xx 错误则发生在服务端存在问题时。所以，我们只需要确保 download 函数在发生 5xx 错误时重试下载即可。下面是支持重试下载功能的新版本代码。

```
def download(url, num_retries=2):
    print 'Downloading:', url
    try:
        html = urllib2.urlopen(url).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
        html = None
        if num_retries > 0:
            if hasattr(e, 'code') and 500 <= e.code < 600:
                # recursively retry 5xx HTTP errors
                return download(url, num_retries-1)
    return html
```

现在，当 download 函数遇到 5xx 错误码时，将会递归调用函数自身进行重试。此外，该函数还增加了一个参数，用于设定重试下载的次数，其默认值为两次。我们在这里限制网页下载的尝试次数，是因为服务器错误可能暂时还没有解决。想要测试该函数，可以尝试下载 http://httpstat.us/500，该网址会始终返回 500 错误码。

```
>>> download('http://httpstat.us/500')
Downloading: http://httpstat.us/500
Download error: Internal Server Error
Downloading: http://httpstat.us/500
Download error: Internal Server Error
Downloading: http://httpstat.us/500
Download error: Internal Server Error
```

从上面的返回结果可以看出，download 函数的行为和预期一致，先尝试下载网页，在接收到 500 错误后，又进行了两次重试才放弃。

2. 设置用户代理

默认情况下，urllib2 使用 Python-urllib/2.7 作为用户代理下载网页内容，其中 2.7 是 Python 的版本号。如果能使用可辨识的用户代理则更好，这样可以避免我们的网络爬虫碰到一些问题。此外，也许是因为曾经历过质量不佳的 Python 网络爬虫造成的服务器过载，一些网站还会封禁这个默认的用户代理。比如，在使用 Python 默认用户代理的情况下，访问 http://www.meetup.com/，目前会返回如图 1.3 所示的访问拒绝提示。

图 1.3

因此，为了下载更加可靠，我们需要控制用户代理的设定。下面的代码对 download 函数进行了修改，设定了一个默认的用户代理“wswp”（即 **Web Scraping with Python** 的首字母缩写）。

```
def download(url, user_agent='wswp', num_retries=2):
    print 'Downloading:', url
    headers = {'User-agent': user_agent}
    request = urllib2.Request(url, headers=headers)
    try:
        html = urllib2.urlopen(request).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
        html = None
```

```
            if num_retries > 0:
                if hasattr(e, 'code') and 500 <= e.code < 600:
                    # retry 5XX HTTP errors
                    return download(url, user_agent, num_retries-1)
        return html
```

现在，我们拥有了一个灵活的下载函数，可以在后续示例中得到复用。该函数能够捕获异常、重试下载并设置用户代理。

1.4.2　网站地图爬虫

在第一个简单的爬虫中，我们将使用示例网站 robots.txt 文件中发现的网站地图来下载所有网页。为了解析网站地图，我们将会使用一个简单的正则表达式，从<loc>标签中提取出 URL。而在下一章中，我们将会介绍一种更加健壮的解析方法——**CSS 选择器**。下面是该示例爬虫的代码。

```
def crawl_sitemap(url):
    # download the sitemap file
    sitemap = download(url)
    # extract the sitemap links
    links = re.findall('<loc>(.*?)</loc>', sitemap)
    # download each link
    for link in links:
        html = download(link)
        # scrape html here
        # ...
```

现在，运行网站地图爬虫，从示例网站中下载所有国家页面。

```
>>> crawl_sitemap('http://example.webscraping.com/sitemap.xml')
Downloading: http://example.webscraping.com/sitemap.xml
Downloading: http://example.webscraping.com/view/Afghanistan-1
Downloading: http://example.webscraping.com/view/Aland-Islands-2
Downloading: http://example.webscraping.com/view/Albania-3
...
```

可以看出，上述运行结果和我们的预期一致，不过正如前文所述，我们无法依靠 Sitemap 文件提供每个网页的链接。下一节中，我们将会介绍另一个简单的爬虫，该爬虫不再依赖于 Sitemap 文件。

1.4.3 ID 遍历爬虫

本节中，我们将利用网站结构的弱点，更加轻松地访问所有内容。下面是一些示例国家的 URL。

- `http://example.webscraping.com/view/Afghanistan-1`
- `http://example.webscraping.com/view/Australia-2`
- `http://example.webscraping.com/view/Brazil-3`

可以看出，这些 URL 只在结尾处有所区别，包括国家名（作为页面别名）和 ID。在 URL 中包含页面别名是非常普遍的做法，可以对搜索引擎优化起到帮助作用。一般情况下，Web 服务器会忽略这个字符串，只使用 ID 来匹配数据库中的相关记录。下面我们将其移除，加载 `http://example.webscraping.com/view/1`，测试示例网站中的链接是否仍然可用。测试结果如图 1.4 所示。

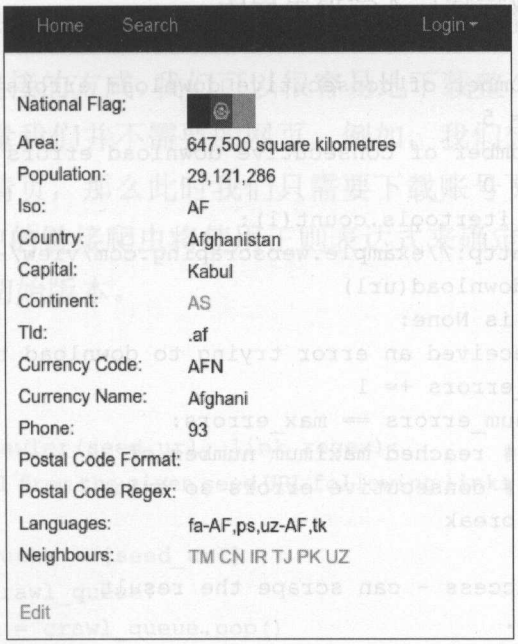


图 1.4

从图 1.4 中可以看出，网页依然可以加载成功，也就是说该方法是有用的。现在，我们就可以忽略页面别名，只遍历 ID 来下载所有国家的页面。下面是使用了该技巧的代码片段。

```
import itertools
for page in itertools.count(1):
    url = 'http://example.webscraping.com/view/-%d' % page
    html = download(url)
    if html is None:
        break
    else:
        # success - can scrape the result
        pass
```

在这段代码中，我们对 ID 进行遍历，直到出现下载错误时停止，我们假设此时已到达最后一个国家的页面。不过，这种实现方式存在一个缺陷，那就是某些记录可能已被删除，数据库 ID 之间并不是连续的。此时，只要访问到某个间隔点，爬虫就会立即退出。下面是这段代码的改进版本，在该版本中连续发生多次下载错误后才会退出程序。

```
# maximum number of consecutive download errors allowed
max_errors = 5
# current number of consecutive download errors
num_errors = 0
for page in itertools.count(1):
    url = 'http://example.webscraping.com/view/-%d' % page
    html = download(url)
    if html is None:
        # received an error trying to download this webpage
        num_errors += 1
        if num_errors == max_errors:
            # reached maximum number of
            # consecutive errors so exit
            break
    else:
        # success - can scrape the result
        # ...
        num_errors = 0
```

上面代码中实现的爬虫需要连续 5 次下载错误才会停止遍历，这样就很大程度上降低了遇到被删除记录时过早停止遍历的风险。

在爬取网站时，遍历 ID 是一个很便捷的方法，但是和网站地图爬虫一样，这种方法也无法保证始终可用。比如，一些网站会检查页面别名是否满足预期，如果不是，则会返回 `404 Not Found` 错误。而另一些网站则会使用非连续大数作为 ID，或是不使用数值作为 ID，此时遍历就难以发挥其作用了。例如，Amazon 使用 ISBN 作为图书 ID，这种编码包含至少 10 位数字。使用 ID 对 Amazon 的图书进行遍历需要测试数十亿次，因此这种方法肯定不是抓取该站内容最高效的方法。

1.4.4 链接爬虫

到目前为止，我们已经利用示例网站的结构特点实现了两个简单爬虫，用于下载所有的国家页面。只要这两种技术可用，就应当使用其进行爬取，因为这两种方法最小化了需要下载的网页数量。不过，对于另一些网站，我们需要让爬虫表现得更像普通用户，跟踪链接，访问感兴趣的内容。

通过跟踪所有链接的方式，我们可以很容易地下载整个网站的页面。但是，这种方法会下载大量我们并不需要的网页。例如，我们想要从一个在线论坛中抓取用户账号详情页，那么此时我们只需要下载账号页，而不需要下载讨论贴的页面。本节中的链接爬虫将使用正则表达式来确定需要下载哪些页面。下面是这段代码的初始版本。

```
import re

def link_crawler(seed_url, link_regex):
    """Crawl from the given seed URL following links matched by link_regex
    """
    crawl_queue = [seed_url]
    while crawl_queue:
        url = crawl_queue.pop()
        html = download(url)
```

```
            # filter for links matching our regular expression
            for link in get_links(html):
                if re.match(link_regex, link):
                    crawl_queue.append(link)

def get_links(html):
    """Return a list of links from html
    """
    # a regular expression to extract all links from the webpage
    webpage_regex = re.compile('<a[^>]+href=["\'](.*?)["\']',
        re.IGNORECASE)
    # list of all links from the webpage
    return webpage_regex.findall(html)
```

要运行这段代码，只需要调用 link_crawler 函数，并传入两个参数：要爬取的网站 URL 和用于跟踪链接的正则表达式。对于示例网站，我们想要爬取的是国家列表索引页和国家页面。其中，索引页链接格式如下。

- http://example.webscraping.com/index/1
- http://example.webscraping.com/index/2

国家页链接格式如下。

- http://example.webscraping.com/view/Afghanistan-1
- http://example.webscraping.com/view/Aland-Islands-2

因此，我们可以用/(index|view)/这个简单的正则表达式来匹配这两类网页。当爬虫使用这些输入参数运行时会发生什么呢？你会发现我们得到了如下的下载错误。

```
>>> link_crawler('http://example.webscraping.com',
    '/(index|view)')
Downloading: http://example.webscraping.com
Downloading: /index/1
Traceback (most recent call last):
    ...
ValueError: unknown url type: /index/1
```

可以看出，问题出在下载/index/1 时，该链接只有网页的路径部分，而没有协议和服务器部分，也就是说这是一个**相对链接**。由于浏览器知道你正在浏览哪个网页，所以在浏览器浏览时，相对链接是能够正常工作的。但是，urllib2 是无法获知上下文的。为了让 urllib2 能够定位网页，我们需要将链接转换为**绝对链接**的形式，以便包含定位网页的所有细节。如你所愿，Python 中确实有用来实现这一功能的模块，该模块称为 urlparse。下面是 link_crawler 的改进版本，使用了 urlparse 模块来创建绝对路径。

```
import urlparse
def link_crawler(seed_url, link_regex):
    """Crawl from the given seed URL following links matched by link_regex
    """
    crawl_queue = [seed_url]
    while crawl_queue:
        url = crawl_queue.pop()
        html = download(url)
        for link in get_links(html):
            if re.match(link_regex, link):
                link = urlparse.urljoin(seed_url, link)
                crawl_queue.append(link)
```

当你运行这段代码时，会发现虽然网页下载没有出现错误，但是同样的地点总是会被不断下载到。这是因为这些地点相互之间存在链接。比如，澳大利亚链接到了南极洲，而南极洲也存在到澳大利亚的链接，此时爬虫就会在它们之间不断循环下去。要想避免重复爬取相同的链接，我们需要记录哪些链接已经被爬取过。下面是修改后的 link_crawler 函数，已具备存储已发现 URL 的功能，可以避免重复下载。

```
def link_crawler(seed_url, link_regex):
    crawl_queue = [seed_url]
    # keep track which URL's have seen before
    seen = set(crawl_queue)
    while crawl_queue:
        url = crawl_queue.pop()
        html = download(url)
        for link in get_links(html):
```

```
            # check if link matches expected regex
            if re.match(link_regex, link):
                # form absolute link
                link = urlparse.urljoin(seed_url, link)
                # check if have already seen this link
                if link not in seen:
                    seen.add(link)
                    crawl_queue.append(link)
```

当运行该脚本时，它会爬取所有地点，并且能够如期停止。最终，我们得到了一个可用的爬虫！

高级功能

现在，让我们为链接爬虫添加一些功能，使其在爬取其他网站时更加有用。

解析 robots.txt

首先，我们需要解析 robots.txt 文件，以避免下载禁止爬取的 URL。使用 Python 自带的 robotparser 模块，就可以轻松完成这项工作，如下面的代码所示。

```
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url('http://example.webscraping.com/robots.txt')
>>> rp.read()
>>> url = 'http://example.webscraping.com'
>>> user_agent = 'BadCrawler'
>>> rp.can_fetch(user_agent, url)
False
>>> user_agent = 'GoodCrawler'
>>> rp.can_fetch(user_agent, url)
True
```

robotparser 模块首先加载 robots.txt 文件，然后通过 can_fetch() 函数确定指定的用户代理是否允许访问网页。在本例中，当用户代理设置为 'BadCrawler' 时，robotparser 模块会返回结果表明无法获取网页，这和示例网站 robots.txt 的定义一样。

为了将该功能集成到爬虫中，我们需要在 crawl 循环中添加该检查。

```
...
while crawl_queue:
    url = crawl_queue.pop()
    # check url passes robots.txt restrictions
    if rp.can_fetch(user_agent, url):
        ...
    else:
        print 'Blocked by robots.txt:', url
```

支持代理

有时我们需要使用代理访问某个网站。比如，Netflix 屏蔽了美国以外的大多数国家。使用 urllib2 支持代理并没有想象中那么容易（可以尝试使用更友好的 Python HTTP 模块 requests 来实现该功能，其文档地址为 http://docs.python-requests.org/）。下面是使用 urllib2 支持代理的代码。

```
proxy = ...
opener = urllib2.build_opener()
proxy_params = {urlparse.urlparse(url).scheme: proxy}
opener.add_handler(urllib2.ProxyHandler(proxy_params))
response = opener.open(request)
```

下面是集成了该功能的新版本 download 函数。

```
def download(url, user_agent='wswp', proxy=None, num_retries=2):
    print 'Downloading:', url
    headers = {'User-agent': user_agent}
    request = urllib2.Request(url, headers=headers)

    opener = urllib2.build_opener()
    if proxy:
        proxy_params = {urlparse.urlparse(url).scheme: proxy}
        opener.add_handler(urllib2.ProxyHandler(proxy_params))
    try:
        html = opener.open(request).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
```

```
            html = None
            if num_retries > 0:
                if hasattr(e, 'code') and 500 <= e.code < 600:
                # retry 5XX HTTP errors
                html = download(url, user_agent, proxy,
                    num_retries-1)
        return html
```

下载限速

如果我们爬取网站的速度过快，就会面临被封禁或是造成服务器过载的风险。为了降低这些风险，我们可以在两次下载之间添加延时，从而对爬虫限速。下面是实现了该功能的类的代码。

```
class Throttle:
    """Add a delay between downloads to the same domain
    """
    def __init__(self, delay):
        # amount of delay between downloads for each domain
        self.delay = delay
        # timestamp of when a domain was last accessed
        self.domains = {}

    def wait(self, url):
        domain = urlparse.urlparse(url).netloc
        last_accessed = self.domains.get(domain)

        if self.delay > 0 and last_accessed is not None:
            sleep_secs = self.delay - (datetime.datetime.now() -
                last_accessed).seconds
            if sleep_secs > 0:
                # domain has been accessed recently
                # so need to sleep
                time.sleep(sleep_secs)
        # update the last accessed time
        self.domains[domain] = datetime.datetime.now()
```

Throttle 类记录了每个域名上次访问的时间，如果当前时间距离上次访问时间小于指定延时，则执行睡眠操作。我们可以在每次下载之前调用 Throttle 对爬虫进行限速。

```
throttle = Throttle(delay)
...
throttle.wait(url)
result = download(url, headers, proxy=proxy,
    num_retries=num_retries)
```

避免爬虫陷阱

目前，我们的爬虫会跟踪所有之前没有访问过的链接。但是，一些网站会动态生成页面内容，这样就会出现无限多的网页。比如，网站有一个在线日历功能，提供了可以访问下个月和下一年的链接，那么下个月的页面中同样会包含访问再下个月的链接，这样页面就会无止境地链接下去。这种情况被称为**爬虫陷阱**。

想要避免陷入爬虫陷阱，一个简单的方法是记录到达当前网页经过了多少个链接，也就是深度。当到达最大深度时，爬虫就不再向队列中添加该网页中的链接了。要实现这一功能，我们需要修改 seen 变量。该变量原先只记录访问过的网页链接，现在修改为一个字典，增加了页面深度的记录。

```
def link_crawler(..., max_depth=2):
    max_depth = 2
    seen = {}
    ...
    depth = seen[url]
    if depth != max_depth:
        for link in links:
            if link not in seen:
                seen[link] = depth + 1
                crawl_queue.append(link)
```

现在有了这一功能，我们就有信心爬虫最终一定能够完成。如果想要禁用该功能，只需将 max_depth 设为一个负数即可，此时当前深度永远不会与之相等。

最终版本

这个高级链接爬虫的完整源代码可以在 https://bitbucket.org/wswp/code/src/tip/chapter01/link_crawler3.py 下载得到。要测

试这段代码，我们可以将用户代理设置为 BadCrawler，也就是本章前文所述的被 robots.txt 屏蔽了的那个用户代理。从下面的运行结果中可以看出，爬虫果然被屏蔽了，代码启动后马上就会结束。

```
>>> seed_url = 'http://example.webscraping.com/index'
>>> link_regex = '/(index|view)'
>>> link_crawler(seed_url, link_regex, user_agent='BadCrawler')
Blocked by robots.txt: http://example.webscraping.com/
```

现在，让我们使用默认的用户代理，并将最大深度设置为 1，这样只有主页上的链接才会被下载。

```
>>> link_crawler(seed_url, link_regex, max_depth=1)
Downloading: http://example.webscraping.com//index
Downloading: http://example.webscraping.com/index/1
Downloading: http://example.webscraping.com/view/Antigua-and-Barbuda-10
Downloading: http://example.webscraping.com/view/Antarctica-9
Downloading: http://example.webscraping.com/view/Anguilla-8
Downloading: http://example.webscraping.com/view/Angola-7
Downloading: http://example.webscraping.com/view/Andorra-6
Downloading: http://example.webscraping.com/view/American-Samoa-5
Downloading: http://example.webscraping.com/view/Algeria-4
Downloading: http://example.webscraping.com/view/Albania-3
Downloading: http://example.webscraping.com/view/Aland-Islands-2
Downloading: http://example.webscraping.com/view/Afghanistan-1
```

和预期一样，爬虫在下载完国家列表的第一页之后就停止了。

1.5　本章小结

本章介绍了网络爬虫，然后开发了一个能够在后续章节中复用的成熟爬虫。此外，我们还介绍了一些外部工具和模块的使用方法，用于了解网站、用户代理、网站地图、爬取延时以及各种爬取策略。

下一章中，我们将讨论如何从已爬取到的网页中获取数据。

第 2 章
数据抓取

在上一章中，我们构建了一个爬虫，可以通过跟踪链接的方式下载我们所需的网页。虽然这个例子很有意思，却不够实用，因为爬虫在下载网页之后又将结果丢弃掉了。现在，我们需要让这个爬虫从每个网页中抽取一些数据，然后实现某些事情，这种做法也被称为**抓取（scraping）**。

首先，我们会介绍一个叫做 **Firebug Lite** 的浏览器扩展，用于检查网页内容，如果你有一些网络开发背景的话，可能已经对该扩展十分熟悉了。然后，我们会介绍三种抽取网页数据的方法，分别是正则表达式、Beautiful Soup 和 lxml。最后，我们将对比这三种数据抓取方法。

2.1 分析网页

想要了解一个网页的结构如何，可以使用查看源代码的方法。在大多数浏览器中，都可以在页面上右键单击选择 **View page source** 选项，获取网页的源代码，如图 2.1 所示。

我们可以在 HTML 的下述代码中找到我们感兴趣的数据。

```
<table>
<tr id="places_national_flag__row"><td class="w2p_fl"><label
  for="places_national_flag"
```

```
        id="places_national_flag__label">National Flag:
          </label></td><td class="w2p_fw"><img
            src="/places/static/images/flags/gb.png" /></td><td
              class="w2p_fc"></td></tr>
    ...
    <tr id="places_neighbours__row"><td class="w2p_fl"><label
      for="places_neighbours"
        id="places_neighbours__label">Neighbours: </label></td><td
          class="w2p_fw"><div><a href="/iso/IE">IE </a></div></td><td
            class="w2p_fc"></td></tr></table>
```

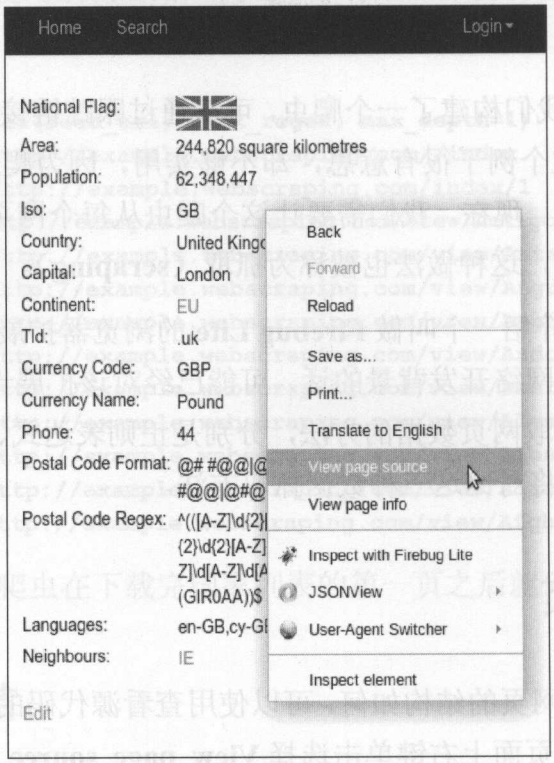


图 2.1

对于浏览器解析而言，缺失空白符和格式并无大碍，但在我们阅读时则会造成一定困难。要想更好地理解该表格，我们将使用 Firebug Lite 扩展。该扩展适用于所有浏览器，我们可以通过 `https://getfirebug.com/firebuglite` 页面获取到该扩展。如果愿意的话，Firefox 用户可以安装完整版的 Firebug

扩展，不过 Lite 版本已经包含了我们在本章和第 6 章中所用到的功能。

Firebug Lite 安装完成后，可以右键单击我们在抓取中感兴趣的网页部分，然后在菜单中选择 **Inspect with Firebug Lite**，如图 2.2 所示。

图 2.2

此时，浏览器就会打开如图 2.3 所示的 Firebug 面板，并显示选中元素周围的 HTML 层次结构。

如图 2.3 所示，当选择国家面积这一属性时，我们可以从 Firebug 面板中清晰地看到，该值包含在 class 为 `w2p_fw` 的`<td>`元素中，而`<td>`元素又是 ID 为 `places_area__row` 的`<tr>`元素的子元素。现在，我们就获取到需要抓取的面积数据的所有信息了。

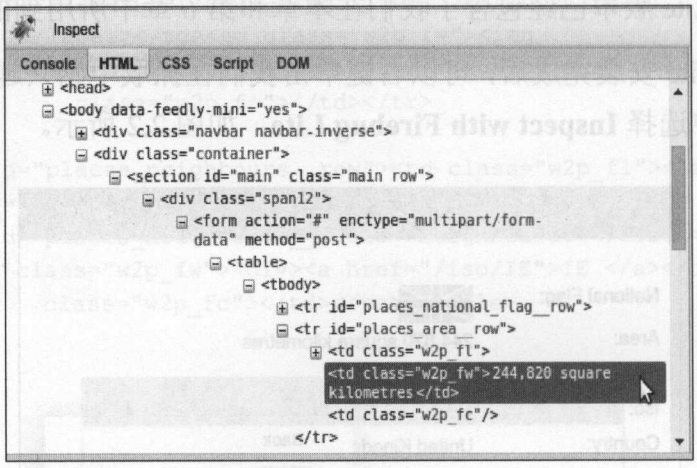


图 2.3

2.2　三种网页抓取方法

现在我们已经了解了该网页的结构，下面将要介绍三种抓取其中数据的方法。首先是正则表达式，然后是流行的 BeautifulSoup 模块，最后是强大的 lxml 模块。

2.2.1　正则表达式

如果你对正则表达式还不熟悉，或是需要一些提示时，可以查阅 https://docs.python.org/2/howto/regex.html 获得完整介绍。

当我们使用正则表达式抓取面积数据时，首先需要尝试匹配<td>元素中的内容，如下所示。

```
>>> import re
>>> url = 'http://example.webscraping.com/view/United
        -Kingdom-239'
>>> html = download(url)
>>> re.findall('<td class="w2p_fw">(.*?)</td>', html)
['<img src="/places/static/images/flags/gb.png" />',
```

```
    '244,820 square kilometres',
    '62,348,447',
    'GB',
    'United Kingdom',
    'London',
    '<a href="/continent/EU">EU</a>',
    '.uk',
    'GBP',
    'Pound',
    '44',
    '@# #@@|@## #@@|@@# #@@|@@## #@@|@#@ #@@|@@#@ #@@|GIR0AA',
    '^(([A-Z]\\d{2}[A-Z]{2})|([A-Z]\\d{3}[A-Z]{2})|([A-Z]{2}\\d{2}
        [A-Z]{2})|([A-Z]{2}\\d{3}[A-Z]{2})|([A-Z]\\d[A-Z]\\d[A-Z]{2})
          |([A-Z]{2}\\d[A-Z]\\d[A-Z]{2})|(GIR0AA))$',
    'en-GB,cy-GB,gd',
    '<div><a href="/iso/IE">IE </a></div>']
```

从上述结果中可以看出，多个国家属性都使用了<td class="w2p_fw">标签。要想分离出面积属性，我们可以只选择其中的第二个元素，如下所示。

```
>>> re.findall('<td class="w2p_fw">(.*?)</td>', html)[1]
'244,820 square kilometres'
```

虽然现在可以使用这个方案，但是如果网页发生变化，该方案很可能就会失效。比如表格发生了变化，去除了第二行中的国土面积数据。如果我们只在现在抓取数据，就可以忽略这种未来可能发生的变化。但是，如果我们希望未来还能再次抓取该数据，就需要给出更加健壮的解决方案，从而尽可能避免这种布局变化所带来的影响。想要该正则表达式更加健壮，我们可以将其父元素<tr>也加入进来。由于该元素具有 ID 属性，所以应该是唯一的。

```
>>> re.findall('<tr id="places_area__row"><td
    class="w2p_fl"><label for="places_area"
        id="places_area__label">Area: </label></td><td
          class="w2p_fw">(.*?)</td>', html)
['244,820 square kilometres']
```

这个迭代版本看起来更好一些，但是网页更新还有很多其他方式，同样可以让该正则表达式无法满足。比如，将双引号变为单引号，<td>标签之间添加多余的空格，或是变更 area_label 等。下面是尝试支持这些可能性的改进版本。

```
>>> re.findall('<tr
  id="places_area__row">.*?<td\s*class=["\']w2p_fw["\']>(.*?)
    </td>', html)
['244,820 square kilometres']
```

虽然该正则表达式更容易适应未来变化，但又存在难以构造、可读性差的问题。此外，还有一些微小的布局变化也会使该正则表达式无法满足，比如在<td>标签里添加 title 属性。

从本例中可以看出，正则表达式为我们提供了抓取数据的快捷方式，但是该方法过于脆弱，容易在网页更新后出现问题。幸好，还有一些更好的解决方案，我们会在接下来的小节中继续介绍。

2.2.2　Beautiful Soup

Beautiful Soup 是一个非常流行的 Python 模块。该模块可以解析网页，并提供定位内容的便捷接口。如果你还没有安装该模块，可以使用下面的命令安装其最新版本：

```
pip install beautifulsoup4
```

使用 Beautiful Soup 的第一步是将已下载的 HTML 内容解析为 soup 文档。由于大多数网页都不具备良好的 HTML 格式，因此 Beautiful Soup 需要对其实际格式进行确定。例如，在下面这个简单网页的列表中，存在属性值两侧引号缺失和标签未闭合的问题。

```
<ul class=country>
    <li>Area
    <li>Population
</ul>
```

如果 Population 列表项被解析为 Area 列表项的子元素，而不是并列的两个列表项的话，我们在抓取时就会得到错误的结果。下面让我们看一下 Beautiful Soup 是如何处理的[①]。

```
>>> from bs4 import BeautifulSoup
>>> broken_html = '<ul class=country><li>Area<li>Population</ul>'
>>> # parse the HTML
>>> soup = BeautifulSoup(broken_html, 'html.parser')
>>> fixed_html = soup.prettify()
>>> print fixed_html
<html>
    <body>
        <ul class="country">
            <li>Area</li>
            <li>Population</li>
        </ul>
    </body>
</html>
```

从上面的执行结果中可以看出，Beautiful Soup 能够正确解析缺失的引号并闭合标签，此外还添加了<html>和<body>标签使其成为完整的 HTML 文档。现在可以使用 find()和 find_all()方法来定位我们需要的元素了。

```
>>> ul = soup.find('ul', attrs={'class':'country'})
>>> ul.find('li') # returns just the first match
<li>Area</li>
>>> ul.find_all('li') # returns all matches
[<li>Area</li>, <li>Population</li>]
```

想要了解全部方法和参数，可以查阅 BeautifulSoup 的官方文档，其网址为：http://www.crummy.com/software/BeautifulSoup/bs4/doc/。

① 译者注：此处使用的是 Python 内置库，由于不同版本的容错能力有所区别，读者在实践中可能无法得到正确结果。相关内容可参考：https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser。

下面是使用该方法抽取示例国家面积数据的完整代码。

```
>>> from bs4 import BeautifulSoup
>>> url = 'http://example.webscraping.com/places/view/
        United-Kingdom-239'
>>> html = download(url)
>>> soup = BeautifulSoup(html)
>>> # locate the area row
>>> tr = soup.find(attrs={'id':'places_area__row'})
>>> td = tr.find(attrs={'class':'w2p_fw'}) # locate the area tag
>>> area = td.text # extract the text from this tag
>>> print area
244,820 square kilometres
```

这段代码虽然比正则表达式的代码更加复杂，但更容易构造和理解。而且，像多余的空格和标签属性这种布局上的小变化，我们也无须再担心了。

2.2.3　Lxml

Lxml 是基于 `libxml2` 这一 XML 解析库的 Python 封装。该模块使用 C 语言编写，解析速度比 Beautiful Soup 更快，不过安装过程也更为复杂。最新的安装说明可以参考 `http://Lxml.de/installation.html`。

和 Beautiful Soup 一样，使用 `lxml` 模块的第一步也是将有可能不合法的 HTML 解析为统一格式。下面是使用该模块解析同一个不完整 HTML 的例子。

```
>>> import lxml.html
>>> broken_html = '<ul class=country><li>Area<li>Population</ul>'
>>> tree = lxml.html.fromstring(broken_html) # parse the HTML
>>> fixed_html = lxml.html.tostring(tree, pretty_print=True)
>>> print fixed_html
<ul class="country">
    <li>Area</li>
    <li>Population</li>
</ul>
```

同样地，`lxml` 也可以正确解析属性两侧缺失的引号，并闭合标签，不过该模块没有额外添加<html>和<body>标签。

解析完输入内容之后，进入选择元素的步骤，此时 lxml 有几种不同的方法，比如 XPath 选择器和类似 Beautiful Soup 的 find() 方法。不过，在本例和后续示例中，我们将会使用 CSS 选择器，因为它更加简洁，并且能够在第 5 章解析动态内容时得以复用。此外，一些拥有 jQuery 选择器相关经验的读者也会对其更加熟悉。

下面是使用 lxml 的 CSS 选择器抽取面积数据的示例代码。

```
>>> tree = lxml.html.fromstring(html)
>>> td = tree.cssselect('tr#places_area__row > td.w2p_fw')[0]
>>> area = td.text_content()
>>> print area
244,820 square kilometres
```

CSS 选择器的关键代码行已被加粗显示。该行代码首先会找到 ID 为 places_area__row 的表格行元素，然后选择 class 为 w2p_fw 的表格数据子标签。

CSS 选择器

CSS 选择器表示选择元素所使用的模式。下面是一些常用的选择器示例。

```
选择所有标签: *
选择<a>标签: a
选择所有 class="link"的元素: .link
选择 class="link"的<a>标签: a.link
选择 id="home"的<a>标签: a#home
选择父元素为<a>标签的所有<span>子标签: a > span
选择<a>标签内部的所有<span>标签: a span
选择 title 属性为"Home"的所有<a>标签: a[title=Home]
```

W3C 已提出 CSS3 规范，其网址为 http://www.w3.org/TR/2011/REC-css3-selectors-20110929/。

Lxml 已经实现了大部分 CSS3 属性，其不支持的功能可以参见 https://pythonhosted.org/cssselect/#supported-selectors。

需要注意的是，lxml 在内部实现中，实际上是将 CSS 选择器转换为等价的 XPath 选择器。

2.2.4　性能对比

要想更好地对本章中介绍的三种抓取方法评估取舍，我们需要对其相对效率进行对比。一般情况下，爬虫会抽取网页中的多个字段。因此，为了让对比更加真实，我们将为本章中的每个爬虫都实现一个扩展版本，用于抽取国家网页中的每个可用数据。首先，我们需要回到 Firebug 中，检查国家页面其他特征的格式，如图 2.4 所示。

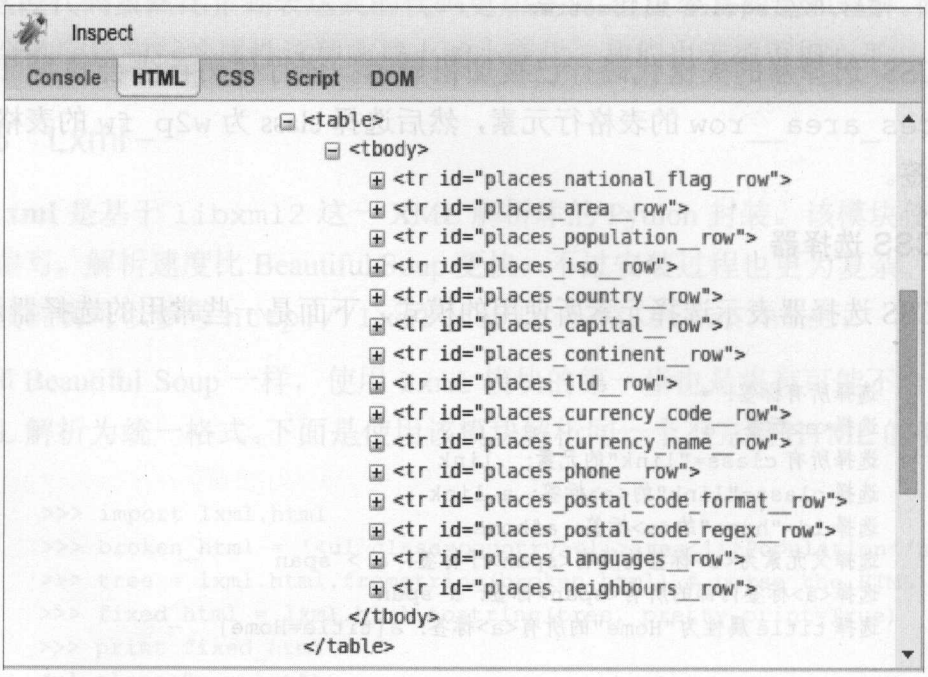


图 2.4

从 Firebug 的显示中可以看出，表格中的每一行都拥有一个以 places_ 起始且以 __row 结束的 ID。而在这些行中包含的国家数据，其格式都和上面的例子相同。下面是使用上述信息抽取所有可用国家数据的实现代码。

```
FIELDS = ('area', 'population', 'iso', 'country', 'capital',
  'continent', 'tld', 'currency_code', 'currency_name', 'phone',
    'postal_code_format', 'postal_code_regex', 'languages',
      'neighbours')

import re
def re_scraper(html):
    results = {}
    for field in FIELDS:
        results[field] = re.search('<tr id="places_%s__row">.*?<td
            class="w2p_fw">(.*?)</td>' % field, html).groups()[0]
    return results

from bs4 import BeautifulSoup
def bs_scraper(html):
    soup = BeautifulSoup(html, 'html.parser')
    results = {}
    for field in FIELDS:
        results[field] = soup.find('table').find('tr',
            id='places_%s__row' % field).find('td',
                class_='w2p_fw').text
    return results

import lxml.html
def lxml_scraper(html):
    tree = lxml.html.fromstring(html)
    results = {}
    for field in FIELDS:
        results[field] = tree.cssselect('table > tr#places_%s__row
            > td.w2p_fw' % field)[0].text_content()
    return results
```

抓取结果

现在，我们已经完成了所有爬虫的代码实现，接下来将通过如下代码片段，测试这三种方法的相对性能。

```
import time
NUM_ITERATIONS = 1000 # number of times to test each scraper
html = download('http://example.webscraping.com/places/view/
  United-Kingdom-239')
```

```
    for name, scraper in [('Regular expressions', re_scraper),
    ('BeautifulSoup', bs_scraper),
    ('Lxml', lxml_scraper)]:
        # record start time of scrape
        start = time.time()
        for i in range(NUM_ITERATIONS):
            if scraper == re_scraper:
                re.purge()
            result = scraper(html)
            # check scraped result is as expected
            assert(result['area'] == '244,820 square kilometres')
        # record end time of scrape and output the total
        end = time.time()
        print '%s: %.2f seconds' % (name, end - start)
```

在这段代码中，每个爬虫都会执行 1000 次，每次执行都会检查抓取结果是否正确，然后打印总用时。这里使用的 download 函数依然是上一章中定义的那个函数。请注意，我们在加粗的代码行中调用了 re.purge() 方法。默认情况下，正则表达式模块会缓存搜索结果，为了与其他爬虫的对比更加公平，我们需要使用该方法清除缓存。

下面是在我的电脑中运行该脚本的结果。

```
$ python performance.py
Regular expressions: 5.50 seconds
BeautifulSoup: 42.84 seconds
Lxml: 7.06 seconds
```

由于硬件条件的区别，不同电脑的执行结果也会存在一定差异。不过，每种方法之间的相对差异应当是相当的。从结果中可以看出，在抓取我们的示例网页时，Beautiful Soup 比其他两种方法慢了超过 6 倍之多。实际上这一结果是符合预期的，因为 lxml 和正则表达式模块都是 C 语言编写的，而 BeautifulSoup 则是纯 Python 编写的。一个有趣的事实是，lxml 表现得和正则表达式差不多好。由于 lxml 在搜索元素之前，必须将输入解析为内部格式，因此会产生额外的开销。而当抓取同一网页的多个特征时，这种初始化解析产生的开销就会降低，lxml 也就更具竞争力。这真是一个令人惊叹的模块！

2.2.5 结论

表 2.1 总结了每种抓取方法的优缺点。

表 2.1

抓取方法	性能	使用难度	安装难度
正则表达式	快	困难	简单（内置模块）
Beautiful Soup	慢	简单	简单（纯 Python）
Lxml	快	简单	相对困难

如果你的爬虫瓶颈是下载网页，而不是抽取数据的话，那么使用较慢的方法（如 Beautiful Soup）也不成问题。如果只需抓取少量数据，并且想要避免额外依赖的话，那么正则表达式可能更加适合。不过，通常情况下，`lxml` 是抓取数据的最好选择，这是因为该方法既快速又健壮，而正则表达式和 Beautiful Soup 只在某些特定场景下有用。

2.2.6 为链接爬虫添加抓取回调

前面我们已经了解了如何抓取国家数据，接下来我们需要将其集成到上一章的链接爬虫当中。要想复用这段爬虫代码抓取其他网站，我们需要添加一个 `callback` 参数处理抓取行为。`callback` 是一个函数，在发生某个特定事件之后会调用该函数（在本例中，会在网页下载完成后调用）。该抓取 `callback` 函数包含 `url` 和 `html` 两个参数，并且可以返回一个待爬取的 URL 列表。下面是其实现代码，可以看出在 Python 中实现该功能非常简单。

```
def link_crawler(..., scrape_callback=None):
    ...
    links = []
    if scrape_callback:
        links.extend(scrape_callback(url, html) or [])
        ...
```

在上面的代码片段中，我们加粗显示了新增加的抓取 callback 函数代码。如果想要获取该版本链接爬虫的完整代码，可以访问 https://bitbucket.org/wswp/code/src/tip/chapter02/link_crawler.py。

现在，我们只需对传入的 scrape_callback 函数定制化处理，就能使用该爬虫抓取其他网站了。下面对 lxml 抓取示例的代码进行了修改，使其能够在 callback 函数中使用。

```
def scrape_callback(url, html):
    if re.search('/view/', url):
        tree = lxml.html.fromstring(html)
        row = [tree.cssselect('table > tr#places_%s__row >
            td.w2p_fw' % field)[0].text_content() for field in
                FIELDS]
        print url, row
```

上面这个 callback 函数会去抓取国家数据，然后将其显示出来。不过通常情况下，在抓取网站时，我们更希望能够复用这些数据，因此下面我们对其功能进行扩展，把得到的结果数据保存到 CSV 表格中，其代码如下所示。

```
import csv
class ScrapeCallback:
    def __init__(self):
        self.writer = csv.writer(open('countries.csv', 'w'))
        self.fields = ('area', 'population', 'iso', 'country',
         'capital', 'continent', 'tld', 'currency_code',
            'currency_name', 'phone', 'postal_code_format',
                'postal_code_regex', 'languages',
                    'neighbours')
        self.writer.writerow(self.fields)

    def __call__(self, url, html):
        if re.search('/view/', url):
            tree = lxml.html.fromstring(html)
            row = []
            for field in self.fields:
                row.append(tree.cssselect('table >
                    tr#places_{}__row >
```

```
                td.w2p_fw'.format(field))
                    [0].text_content())
        self.writer.writerow(row)
```

为了实现该 callback，我们使用了回调类，而不再是回调函数，以便保持 csv 中 writer 属性的状态。csv 的 writer 属性在构造方法中进行了实例化处理，然后在__call__方法中执行了多次写操作。请注意，__call__是一个特殊方法，在对象作为函数被调用时会调用该方法，这也是链接爬虫中 cache_callback 的调用方法。也就是说，scrape_callback(url, html)和调用 scrape_callback.__call__(url, html)是等价的。如果想要了解更多有关 Python 特殊类方法的知识，可以参考 https://docs.python.org/2/reference/datamodel.html#special-method-names。

下面是向链接爬虫传入回调的代码写法。

```
link_crawler('http://example.webscraping.com/', '/(index|view)',
    max_depth=-1, scrape_callback=ScrapeCallback())
```

现在，当我们运行这个使用了 callback 的爬虫时，程序就会将结果写入一个 CSV 文件中，我们可以使用类似 Excel 或者 LibreOffice 的应用查看该文件，如图 2.5 所示。

	A	B	C	D	E	F	G	H	I	J	
1	area	population	iso	country	capital	continent	tld	currency_code	currency_name	phone	postal_code_format
2	390,580 square kilometres	11651858	ZW	Zimbabwe	Harare	AF	.zw	ZWL	Dollar	263	
3	752,614 square kilometres	13460305	ZM	Zambia	Lusaka	AF	.zm	ZMW	Kwacha	260	#####
4	527,970 square kilometres	23495361	YE	Yemen	Sanaa	AS	.ye	YER	Rial	967	
5	266,000 square kilometres	273008	EH	Western Sahara	El-Aaiun	AF	.eh	MAD	Dirham	212	
6	274 square kilometres	16025	WF	Wallis and Futuna	Mata Utu	OC	.wf	XPF	Franc	681	#####
7	329,560 square kilometres	89571130	VN	Vietnam	Hanoi	AS	.vn	VND	Dong	84	######
8	912,050 square kilometres	27223228	VE	Venezuela	Caracas	SA	.ve	VEF	Bolivar	58	####
9	0 square kilometres	921	VA	Vatican	Vatican City	EU	.va	EUR	Euro	379	#####
10	12,200 square kilometres	221552	VU	Vanuatu	Port Vila	OC	.vu	VUV	Vatu	678	
11	447,400 square kilometres	27865738	UZ	Uzbekistan	Tashkent	AS	.uz	UZS	Som	998	######
12	176,220 square kilometres	3477000	UY	Uruguay	Montevideo	SA	.uy	UYU	Peso	598	#####
13	0 square kilometres	0	UM	United States Minor Outlying Islands		OC	.um	USD	Dollar	1	
14	9,629,091 square kilometres	310232863	US	United States	Washington	NA	.us	USD	Dollar	1	#####-####
15	244,820 square kilometres	62348447	GB	United Kingdom	London	EU	.uk	GBP	Pound	44	@# #@@|@## #@@|@@# #@@@@
16	82,880 square kilometres	4975593	AE	United Arab Emirates	Abu Dhabi	AS	.ae	AED	Dirham	971	
17	603,700 square kilometres	45415596	UA	Ukraine	Kiev	EU	.ua	UAH	Hryvnia	380	#####
18	236,040 square kilometres	33398682	UG	Uganda	Kampala	AF	.ug	UGX	Shilling	256	
19	352 square kilometres	109708	VI	U.S. Virgin Islands	Charlotte Amalie	NA	.vi	USD	Dollar	+1-340	#####-####
20	26 square kilometres	10472	TV	Tuvalu	Funafuti	OC	.tv	AUD	Dollar	688	
21	430 square kilometres	20556	TC	Turks and Caicos Islands	Cockburn Town	NA	.tc	USD	Dollar	+1-649	TKCA 1ZZ
22	488,100 square kilometres	4940916	TM	Turkmenistan	Ashgabat	AS	.tm	TMT	Manat	993	######
23	780,580 square kilometres	77804122	TR	Turkey	Ankara	AS	.tr	TRY	Lira	90	#####
24	163,610 square kilometres	10589025	TN	Tunisia	Tunis	AF	.tn	TND	Dinar	216	####
25	5,128 square kilometres	1228691	TT	Trinidad and Tobago	Port of Spain	NA	.tt	TTD	Dollar	+1-868	
26	748 square kilometres	122580	TO	Tonga	Nuku'alofa	OC	.to	TOP	Pa'anga	676	
27	10 square kilometres	1466	TK	Tokelau		OC	.tk	NZD	Dollar	690	
28	56,785 square kilometres	6587239	TG	Togo	Lome	AF	.tg	XOF	Franc	228	
29	514,000 square kilometres	67089500	TH	Thailand	Bangkok	AS	.th	THB	Baht	66	#####

countries

图 2.5

成功了！我们完成了第一个可以工作的数据抓取爬虫。

2.3　本章小结

在本章中，我们介绍了几种抓取网页数据的方法。正则表达式在一次性数据抓取中非常有用，此外还可以避免解析整个网页带来的开销；BeautifulSoup 提供了更高层次的接口，同时还能避免过多麻烦的依赖。不过，通常情况下，lxml 是我们的最佳选择，因为它速度更快，功能更加丰富，因此在接下来的例子中我们将会使用 lxml 模块进行数据抓取。

下一章，我们会介绍缓存技术，这样就能把网页保存下来，只在爬虫第一次运行时才会下载网页。

第 3 章
下载缓存

在上一章中，我们学习了如何从已爬取到的网页中抓取数据，以及将抓取结果保存到表格中。如果我们还想抓取另外一个字段，比如国旗图片的URL，那么又该怎么做呢？要想抓取这些新增的字段，我们需要重新下载整个网站。对于我们这个小型的示例网站而言，这可能不算特别大的问题。但是，对于那些拥有数百万个网页的网站而言，重新爬取可能需要耗费几个星期的时间。因此，本章提出了对已爬取网页进行缓存的方案，可以让每个网页只下载一次。

3.1 为链接爬虫添加缓存支持

要想支持缓存，我们需要修改第 1 章中编写的 download 函数，使其在URL 下载之前进行缓存检查。另外，我们还需要把限速功能移至函数内部，只有在真正发生下载时才会触发限速，而在加载缓存时不会触发。为了避免每次下载都要传入多个参数，我们借此机会将 download 函数重构为一个类，这样参数只需在构造方法中设置一次，就能在后续下载时多次复用。下面是支持了缓存功能的代码实现。

```
class Downloader:
    def __init__(self, delay=5,
```

```
        user_agent='wswp', proxies=None,
            num_retries=1, cache=None):
        self.throttle = Throttle(delay)
        self.user_agent = user_agent
        self.proxies = proxies
        self.num_retries = num_retries
        self.cache = cache

    def __call__(self, url):
        result = None
        if self.cache:
            try:
                result = self.cache[url]
            except KeyError:
                # url is not available in cache
                pass
            else:
                if self.num_retries > 0 and \
                    500 <= result['code'] < 600:
                    # server error so ignore result from cache
                    # and re-download
                    result = None
        if result is None:
            # result was not loaded from cache
            # so still need to download
            self.throttle.wait(url)
            proxy = random.choice(self.proxies) if self.proxies
                else None
            headers = {'User-agent': self.user_agent}
            result = self.download(url, headers, proxy,
                self.num_retries)
            if self.cache:
                # save result to cache
                self.cache[url] = result
        return result['html']

def download(self, url, headers, proxy, num_retries,
    data=None):
    ...
return {'html': html, 'code': code}
```

下载类的完整源码可以从 https://bitbucket.org/wswp/code/src/tip/chapter03/downloader.py 获取。

前面代码中的 Download 类有一个比较有意思的部分，那就是__call__特殊方法，在该方法中我们实现了下载前检查缓存的功能。该方法首先会检查缓存是否已经定义。如果已经定义，则检查之前是否已经缓存了该 URL。如果该 URL 已被缓存，则检查之前的下载中是否遇到了服务端错误。最后，如果也没有发生过服务端错误，则表明该缓存结果可用。如果上述检查中的任何一项失败，都需要正常下载该 URL，然后将得到的结果添加到缓存中。这里的 download 方法和之前的 download 函数基本一样，只是在返回下载的 HTML 时额外返回了 HTTP 状态码，以便在缓存中存储错误码。当然，如果你只需要一个简单的下载功能，而不需要限速或缓存的话，可以直接调用该方法，这样就不会通过__call__方法调用了。

而对于 cache 类，我们可以通过调用 result = cache[url]从 cache 中加载数据，并通过 cache[url] = result 向 cache 中保存结果。大家应该很熟悉这种便捷的接口写法，因为这也是 Python 内建字典数据类型的使用方式。为了支持该接口，我们的 cache 类需要定义__getitem__()和__setitem__()这两个特殊的类方法。

此外，为了支持缓存功能，链接爬虫的代码也需要进行一些微调，包括添加 cache 参数、移除限速以及将 download 函数替换为新的类等，如下面的代码所示。

```
def link_crawler(..., cache=None):
    crawl_queue = [seed_url]
    seen = {seed_url: 0}
    num_urls = 0
    rp = get_robots(seed_url)
    D = Downloader(delay=delay, user_agent=user_agent,
        proxies=proxies, num_retries=num_retries, cache=cache)
```

```
while crawl_queue:
    url = crawl_queue.pop()
    depth = seen[url]
    # check url passes robots.txt restrictions
    if rp.can_fetch(user_agent, url):
        html = D(url)
        links = []
        ...
```

到目前为止，这个网络爬虫的基本架构已经准备好了，下面就要开始构建实际的缓存了。

3.2　磁盘缓存

要想缓存下载结果，我们先来尝试最容易想到的方案，将下载到的网页存储到文件系统中。为了实现该功能，我们需要将 URL 安全地映射为跨平台的文件名。表 3.1 所示为几大主流文件系统的限制。

表 3.1

操作系统	文件系统	非法文件名字符	文件名最大长度
Linux	Ext3/Ext4	/ 和 \0	255 字节
OS X	HFS Plus	: 和 \0	255 个 UTF-16 编码单元
Windows	NTFS	\、/、?、:、*、"、>、< 和 |	255 个字符

为了保证在不同文件系统中，我们的文件路径都是安全的，就需要限制其只能包含数字、字母和基本符号，并将其他字符替换为下划线，其实现代码如下所示。

```
>>> import re
>>> url = 'http://example.webscraping.com/default/view/
    Australia-1'
>>> re.sub('[^/0-9a-zA-Z\-.,;_ ]', '_', url)
'http_//example.webscraping.com/default/view/Australia-1'
```

此外，文件名及其父目录的长度需要限制在255个字符以内（实现代码如下），以满足表3.1中给出的长度限制。

```
>>> filename = '/'.join(segment[:255] for segment in
    filename.split('/'))
```

还有一种边界情况需要考虑，那就是URL路径可能会以斜杠（/）结尾，此时斜杠后面的空字符串就会成为一个非法的文件名。但是，如果移除这个斜杠，使用其父字符串作为文件名，又会造成无法保存其他URL的问题。考虑下面这两个URL：

- `http://example.webscraping.com/index/`
- `http://example.webscraping.com/index/1`

如果我们希望这两个URL都能保存下来，就需要以index作为目录名，以1作为子路径。对于像第一个URL路径这样以斜杠结尾的情况，这里使用的解决方案是添加`index.html`作为其文件名。同样地，当URL路径为空时也进行相同的操作。为了解析URL，我们需要使用`urlparse.urlsplit()`函数，将URL分割成几个部分。

```
>>> import urlparse
>>> components =
    urlparse.urlsplit('http://example.webscraping.com/index/')
>>> print components
SplitResult(scheme='http', netloc='example.webscraping.com',
    path='/index/', query='', fragment='')
>>> print components.path
'/index/'
```

该函数提供了解析和处理URL的便捷接口。下面是使用该模块对上述边界情况添加`index.html`的示例代码。

```
>>> path = components.path
>>> if not path:
>>>     path = '/index.html'
>>> elif path.endswith('/'):
```

```
>>>     path += 'index.html'
>>> filename = components.netloc + path + components.query
>>> filename
'example.webscraping.com/index/index.html'
```

3.2.1　实现

上一节中，我们介绍了创建基于磁盘的缓存时需要考虑的文件系统限制，包括允许使用哪些字符、文件名长度限制，以及确保文件和目录的创建位置不同。把 URL 到文件名的这些映射逻辑结合起来，就形成了磁盘缓存的主要部分。下面是 DiskCache 类的初始实现代码。

```
import os
import re
import urlparse

class DiskCache:
    def __init__(self, cache_dir='cache'):
        self.cache_dir = cache_dir
        self.max_length = max_length

    def url_to_path(self, url):
        """Create file system path for this URL
        """
        components = urlparse.urlsplit(url)
        # append index.html to empty paths
        path = components.path
        if not path:
            path = '/index.html'
        elif path.endswith('/'):
            path += 'index.html'
        filename = components.netloc + path + components.query
        # replace invalid characters
        filename = re.sub('[^/0-9a-zA-Z\-.,;_ ]', '_', filename)
        # restrict maximum number of characters
        filename = '/'.join(segment[:255] for segment in
            filename.split('/'))
        return os.path.join(self.cache_dir, filename)
```

在上面的代码中，构造方法传入了一个用于设定缓存位置的参数，然后在 url_to_path 方法中应用了前面讨论的文件名限制。现在，我们还缺少根据文件名存取数据的方法，下面的代码实现了这两个缺失的方法。

```
import pickle
class DiskCache:
    ...
    def __getitem__(self, url):
        """Load data from disk for this URL
        """
        path = self.url_to_path(url)
        if os.path.exists(path):
            with open(path, 'rb') as fp:
                return pickle.load(fp)
        else:
            # URL has not yet been cached
            raise KeyError(url + ' does not exist')

    def __setitem__(self, url, result):
        """Save data to disk for this url
        """
        path = self.url_to_path(url)
        folder = os.path.dirname(path)
        if not os.path.exists(folder):
            os.makedirs(folder)
        with open(path, 'wb') as fp:
            fp.write(pickle.dumps(result))
```

在__setitem()__中，我们使用 url_to_path()方法将 URL 映射为安全文件名，在必要情况下还需要创建父目录。这里使用的 pickle 模块会把输入转化为字符串，然后保存到磁盘中。而在__getitem__()方法中，首先将 URL 映射为安全文件名。然后，如果文件存在，则加载其内容，并执行反序列化，恢复其原始数据类型；如果文件不存在，则说明缓存中还没有该 URL 的数据，此时会抛出 KeyError 异常。

3.2.2　缓存测试

现在，我们通过向爬虫传递 cache 回调，来检验 DiskCache 类。该类的完整源代码可以从 https://bitbucket.org/wswp/code/src/tip/chapter03/disk_cache.py 获取。我们可以通过执行如下脚本，使用链接爬虫测试磁盘缓存。

```
$ time python disk_cache.py
Downloading: http://example.webscraping.com
Downloading: http://example.webscraping.com/view/Afghanistan-1
...
Downloading: http://example.webscraping.com/view/Zimbabwe-252
23m38.289s
```

第一次执行该命令时，由于缓存为空，因此网页会被正常下载。但当我们第二次执行该脚本时，网页加载自缓存中，爬虫应该更快完成执行，其执行结果如下所示。

```
$ time python disk_cache.py
0m0.186s
```

和上面的预期一样，爬取操作很快就完成了。当缓存为空时，我的计算机中的爬虫下载耗时超过 23 分钟；而在第二次全部使用缓存时，该耗时只有 0.186 秒（比第一次爬取快了超过 7000 倍！）。由于硬件的差异，在不同的计算机中的准确执行时间也会有所区别。不过毋庸置疑的是，磁盘缓存速度更快。

3.2.3　节省磁盘空间

为了最小化缓存所需的磁盘空间，我们可以对下载得到的 HTML 文件进行压缩处理。处理的实现方法很简单，只需在保存到磁盘之前使用 zlib 压缩序列化字符串即可，如下面的代码所示。

```
fp.write(zlib.compress(pickle.dumps(result)))
```

而从磁盘加载后解压的代码如下所示。

```
return pickle.loads(zlib.decompress(fp.read()))
```

压缩完所有网页之后，缓存大小从 4.4MB 下降到 2.3MB，而在我的计算机上爬取缓存示例网站的时间是 0.212 秒，和未压缩时的 0.186 秒相比只是略有增加。当然，如果你的项目对速度十分敏感的话，也可以禁用压缩功能。

3.2.4 清理过期数据

当前版本的磁盘缓存使用键值对的形式在磁盘上保存缓存，未来无论何时请求都会返回结果。对于缓存网页而言，该功能可能不太理想，因为网页内容随时都有可能发生变化，存储在缓存中的数据存在过期风险。本节中，我们将为缓存数据添加过期时间，以便爬虫知道何时需要重新下载网页。在缓存网页时支持存储时间戳的功能也很简单，如下面的代码所示。

```
from datetime import datetime, timedelta

class DiskCacke:
    def _init_(self, ..., expires=timedelta(days=30)):
    ...
    self.expires = expires

def _getitem_(self, url):
    """Load data from disk for this URL
    """
    ...
        with open(path, 'rb') as fp:
                result, timestamp =
                    pickle.loads(zlib.decompress(fp.read()))
                if self.has_expired(timestamp):
                    raise KeyError(url + ' has expired')
                return result
        else:
            # URL has not yet been cached
            raise KeyError(url + ' does not exist')

def _setitem_(self, url, result):
```

```
        """Save data to disk for this url
        """
        ...
        timestamp = datetime.utcnow()
        data = pickle.dumps((result, timestamp))
        with open(path, 'wb') as fp:
            fp.write(zlib.compress(data))

    def has_expired(self, timestamp):
        """Return whether this timestamp has expired
        """
        return datetime.utcnow() > timestamp + self.expires
```

在构造方法中，我们使用 timedelta 对象将默认过期时间设置为 30 天。然后，在__set__方法中，把当前时间戳保存到序列化数据中；而在__get__方法中，对比当前时间和缓存时间，检查是否过期。为了测试过期时间功能，我们可以将其缩短为 5 秒，如下所示。

```
>>> cache = DiskCache(expires=timedelta(seconds=5))
>>> url = 'http://example.webscraping.com'
>>> result = {'html': '...'}
>>> cache[url] = result
>>> cache[url]
{'html': '...'}
>>> import time; time.sleep(5)
>>> cache[url]
Traceback (most recent call last):
...
KeyError: 'http://example.webscraping.com has expired'
```

和预期一样，缓存结果最初是可用的，经过 5 秒的睡眠之后，再次调用同一 URL，则会抛出 KeyError 异常，也就是说缓存下载失效了。

3.2.5　缺点

基于磁盘的缓存系统比较容易实现，无须安装其他模块，并且在文件管理器中就能查看结果。但是，该方法存在一个缺点，即受制于本地文件系统的限制。本章早些时候，为了将 URL 映射为安全文件名，我们应用了多种限制，

然而这又会引发另一个问题，那就是一些 URL 会被映射为相同的文件名。比如，在对如下几个 URL 进行字符替换之后就会得到相同的文件名。

- `http://example.com/?a+b`
- `http://example.com/?a*b`
- `http://example.com/?a=b`
- `http://example.com/?a!b`

这就意味着，如果其中一个 URL 生成了缓存，其他 3 个 URL 也会被认为已经生成缓存，因为它们映射到了同一个文件名。另外，如果一些长 URL 只在 255 个字符之后存在区别，截断后的版本也会被映射为相同的文件名。这个问题非常重要，因为 URL 的最大长度并没有明确限制。尽管在实践中 URL 很少会超过 2000 个字符，并且早期版本的 IE 浏览器也不支持超过 2083 个字符的 URL。

避免这些限制的一种解决方案是使用 URL 的哈希值作为文件名。尽管该方法可以带来一定改善，但是最终还是会面临许多文件系统具有的一个关键问题，那就是每个卷和每个目录下的文件数量是有限制的。如果缓存存储在 FAT32 文件系统中，每个目录的最大文件数是 65535。该限制可以通过将缓存分割到不同目录来避免，但是文件系统可存储的文件总数也是有限制的。我使用的 `ext4` 分区目前支持略多于 1500 万个文件，而一个大型网站往往拥有超过 1 亿个网页。很遗憾，`DiskCache` 方法想要通用的话存在太多限制。要想避免这些问题，我们需要把多个缓存网页合并到一个文件中，并使用类似 `B+`树的算法进行索引。我们并不会自己实现这种算法，而是在下一节中介绍已实现这类算法的数据库。

3.3 数据库缓存

为了避免磁盘缓存方案的已知限制，下面我们会在现有数据库系统之上创

建缓存。爬取时，我们可能需要缓存大量数据，但又无须任何复杂的连接操作，因此我们将选用 NoSQL 数据库，这种数据库比传统的关系型数据库更易于扩展。在本节中，我们将会选用目前非常流行的 MongoDB 作为缓存数据库。

3.3.1　NoSQL 是什么

NoSQL 全称为 **Not Only SQL**，是一种相对较新的数据库设计方式。传统的关系模型使用的是固定模式，并将数据分割到各个表中。然而，对于大数据集的情况，数据量太大使其难以存放在单一服务器中，此时就需要扩展到多台服务器。不过，关系模型对于这种扩展的支持并不够好，因为在查询多个表时，数据可能在不同的服务器中。相反，NoSQL 数据库通常是无模式的，从设计之初就考虑了跨服务器无缝分片的问题。在 NoSQL 中，有多种方式可以实现该目标，分别是列数据存储（如 HBase）、键值对存储（如 Redis）、面向文档的数据库（如 MongoDB）以及图形数据库（如 Neo4j）。

3.3.2　安装 MongoDB

MongoDB 可以从 `https://www.mongodb.org/downloads` 下载得到。然后，我们需要使用如下命令额外安装其 Python 封装库。

```
pip install pymongo
```

要想检测安装是否成功，可以使用如下命令在本地启动 MongoDB。

```
$ mongod -dbpath .
```

然后，在 Python 中，使用 MongoDB 的默认端口尝试连接 MongoDB。

```
>>> from pymongo import MongoClient
>>> client = MongoClient('localhost', 27017)
```

3.3.3　MongoDB 概述

下面是通过 MongoDB 存取数据的示例代码。

```
>>> url = 'http://example.webscraping.com/view/United-Kingdom-239'
>>> html = '...'
>>> db = client.cache
>>> db.webpage.insert({'url': url, 'html': html})
ObjectId('5518c0644e0c87444c12a577')
>>> db.webpage.find_one(url=url)
{u'_id': ObjectId('5518c0644e0c87444c12a577'),
u'html': u'...',
u'url': u'http://example.webscraping.com/view/United-Kingdom-239'}
```

上面的例子存在一个问题，那就是如果我们对相同的 URL 插入另一条不同的文档时，MongoDB 会欣然接受并执行这次插入操作，其执行过程如下所示。

```
>>> db.webpage.insert({'url': url, 'html': html})
>>> db.webpage.find(url=url).count()
2
```

此时，同一 URL 下出现了多条记录，但我们只关心最新存储的那条数据。为了避免重复，我们将 ID 设置为 URL，并执行 upsert 操作。该操作表示当记录存在时更新记录，否则插入新记录，其代码如下所示。

```
>>> db.webpage.update({'_id': url}, {'$set': {'html': html}},
    upsert=True)
>>> db.webpage.find_one({'_id': url})
{u'_id': u'http://example.webscraping.com/view/
    United-Kingdom-239', u'html': u'...'}
```

现在，当我们尝试向同一 URL 插入记录时，将会更新其内容，而不是创建冗余的数据，如下面的代码所示。

```
>>> new_html = '<html></html>'
>>> db.webpage.update({'_id': url}, {'$set': {'html': new_
html}}, upsert=True)
>>> db.webpage.find_one({'_id': url})
{u'_id': u'http://example.webscraping.com/view/United-Kingdom-239',
u'html': u'<html></html>'}
>>> db.webpage.find({'_id': url}).count()
1
```

可以看出，在添加了这条记录之后，虽然 HTML 的内容更新了，但该 URL 的记录数仍然是 1。

MongoDB 官方文档可参考 http://docs.mongodb.org/manual/，在该文档中可以找到上述功能及一些其他功能更详细的介绍。

3.3.4　MongoDB 缓存实现

现在我们已经准备好创建基于 MongoDB 的缓存了，这里使用了和之前的 DiskCache 类相同的类接口。

```
from datetime import datetime, timedelta
from pymongo import MongoClient

class MongoCache:
    def __init__(self, client=None, expires=timedelta(days=30)):
        # if a client object is not passed then try
        # connecting to mongodb at the default localhost port
        self.client = MongoClient('localhost', 27017)
            if client is None else client
        # create collection to store cached webpages,
        # which is the equivalent of a table
        # in a relational database
        self.db = client.cache
        # create index to expire cached webpages
        self.db.webpage.create_index('timestamp',
            expireAfterSeconds=expires.total_seconds())

    def __getitem__(self, url):
        """Load value at this URL
        """
        record = self.db.webpage.find_one({'_id': url})
        if record:
            return record['result']
        else:
            raise KeyError(url + ' does not exist')
```

```
    def __setitem__(self, url, result):
        """Save value for this URL
        """
        record = {'result': result, 'timestamp':
                  datetime.utcnow()}
        self.db.webpage.update({'_id': url}, {'$set': record},
                               upsert=True)
```

在上一节讨论如何避免冗余时，你已经见过这里的__getitem__和__setitem__方法的实现了。此外，我们在构造方法中创建了 timestamp 索引。在达到给定时间戳一定秒数之后，MongoDB 的这一便捷功能可以自动删除记录。这样我们就无须再像 DiskCache 类那样，手工检查记录是否仍然有效了。下面我们使用空的 timedelta 对象进行测试，此时记录在创建后就会被立即删除。

```
>>> cache = MongoCache(expires=timedelta())
>>> result = {'html': '...'}
>>> cache[url] = result
>>> cache[url]
{'html': '...'}
```

记录还在这里，看起来好像我们的缓存过期机制没能正常运行。但实际上这是 MongoDB 的运行机制造成的。MongoDB 运行了一个后台任务，每分钟检查一次过期记录，所以此时该记录还没有被删除。让我们再等 1 分钟，就会发现缓存过期机制已经运行成功了。

```
>>> import time; time.sleep(60)
>>> cache[url]
Traceback (most recent call last):
...
KeyError: 'http://example.webscraping.com/view/United-Kingdom-239
    does not exist'
```

这种机制下，MongoDB 缓存无法按照给定时间精确清理过期记录，会存在至多 1 分钟的延时。不过，由于缓存过期时间通常设定为几周或是几个月，所以这个相对较小的延时不会存在太大问题。

3.3.5　压缩

为了使数据库缓存与之前的磁盘缓存功能一致，我们最后还要添加一个功能：**压缩**。其实现方法和磁盘缓存相类似，即序列化数据后使用 zlib 库进行压缩，如下面的代码所示。

```
import pickle
import zlib
from bson.binary import Binary

class MongoCache:
    def __getitem__(self, url):
        record = self.db.webpage.find_one({'_id': url})
        if record:
            return pickle.loads(zlib.decompress(record['result']))
        else:
            raise KeyError(url + ' does not exist')

    def __setitem__(self, url, result):
        record = {
            'result': Binary(zlib.compress(pickle.dumps(result))),
            'timestamp': datetime.utcnow()
        }
        self.db.webpage.update(
            {'_id': url}, {'$set': record}, upsert=True)
```

3.3.6　缓存测试

MongoCache类的源码可以从https://bitbucket.org/wswp/code/src/tip/chapter03/mongo_cache.py获取，和 DiskCache 一样，这里我们依然通过执行该脚本测试链接爬虫。

```
$ time python mongo_cache.py
http://example.webscraping.com
http://example.webscraping.com/view/Afghanistan-1
...
http://example.webscraping.com/view/Zimbabwe-252
23m40.302s
```

```
$ time python mongo_cache.py
0.378s
```

可以看出，加载数据库缓存的时间几乎是加载磁盘缓存的两倍。不过，MongoDB 可以让我们免受文件系统的各种限制，还能在下一章介绍的并发爬虫处理中更加高效。

3.4 本章小结

本章中，我们了解到缓存已下载的网页可以节省时间，并能最小化重新爬取网站所耗费的带宽。缓存的主要缺点是会占用磁盘空间，不过我们可以使用压缩的方式减少空间占用。此外，在类似 MongoDB 等现有数据库的基础之上创建缓存，可以避免文件系统的各种限制。

下一章，我们会为爬虫添加并发下载多个网页的功能，从而使爬虫运行得更快。

第 4 章
并发下载

在之前的章节中，我们的爬虫都是串行下载网页的，只有前一次下载完成之后才会启动新下载。在爬取规模较小的示例网站时，串行下载尚可应对，但面对大型网站时就会显得捉襟见肘了。在爬取拥有 100 万网页的大型网站时，假设我们以每秒一个网页的速度昼夜不停地下载，耗时也要超过 11 天。如果我们可以同时下载多个网页，那么下载时间将会得到显著改善。

本章将介绍使用多线程和多进程这两种下载网页的方式，并将它们与串行下载的性能进行比较。

4.1 100 万个网页

想要测试并发下载的性能，最好要有一个大型的目标网站。为此，本章将使用 Alexa 提供的最受欢迎的 100 万个网站列表，该列表的排名根据安装了 Alexa 工具栏的用户得出。尽管只有少数用户使用了这个浏览器插件，其数据并不权威，但对于我们这个测试来说已经足够了。

我们可以通过浏览 Alexa 网站获取该数据，其网址为 `http://www.alexa.com/topsites`。此外，我们也可以通过 `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip` 直接下载这一列表的压缩文件，这样就不用再去抓取 Alexa 网站的数据了。

4.1.1　解析 Alexa 列表

Alexa 网站列表是以电子表格的形式提供的，表格中包含两列内容，分别是排名和域名，如图 4.1 所示。

	A	B
1	1	google.com
2	2	facebook.com
3	3	youtube.com
4	4	yahoo.com
5	5	baidu.com
6	6	wikipedia.org
7	7	amazon.com
8	8	twitter.com
9	9	taobao.com
10	10	qq.com

图 4.1

抽取数据包含如下 4 个步骤。

1. 下载.zip 文件。
2. 从.zip 文件中提取出 CSV 文件。
3. 解析 CSV 文件。
4. 遍历 CSV 文件中的每一行，从中抽取出域名数据。

下面是实现上述功能的代码。

```
import csv
from zipfile import ZipFile
from StringIO import StringIO
from downloader import Downloader

D = Downloader()
zipped_data = D('http://s3.amazonaws.com/alexa-static/top-1m.csv.zip')
urls = [] # top 1 million URL's will be stored in this list
with ZipFile(StringIO(zipped_data)) as zf:
    csv_filename = zf.namelist()[0]
```

```
        for _, website in csv.reader(zf.open(csv_filename)):
            urls.append('http://' + website)
```

你可能已经注意到，下载得到的压缩数据是在使用 StringIO 封装之后，才传给 ZipFile 的。这是因为 ZipFile 需要一个类似文件的接口，而不是字符串。接下来，我们从文件名列表中提取出 CSV 文件的名称。由于这个 .zip 文件中只包含一个文件，所以我们直接选择第一个文件名即可。然后遍历该 CSV 文件，将第二列中的域名数据添加到 URL 列表中。为了使 URL 合法，我们还会在每个域名前添加 http://协议。

要想在之前开发的爬虫中复用上述功能，还需要修改 scrape_callback 接口。

```
class AlexaCallback:
    def __init__(self, max_urls=1000):
        self.max_urls = max_urls
        self.seed_url = 'http://s3.amazonaws.com/alexa-static/
            top-1m.csv.zip'

    def __call__(self, url, html):
        if url == self.seed_url:
            urls = []
            with ZipFile(StringIO(html)) as zf:
                csv_filename = zf.namelist()[0]
                for _, website in
                    csv.reader(zf.open(csv_filename)):
                    urls.append('http://' + website)
                    if len(urls) == self.max_urls:
                        break
            return urls
```

这里添加了一个新的输入参数 max_urls，用于设定从 Alexa 文件中提取的 URL 数量。默认情况下，该值被设置为 1000 个 URL，这是因为下载 100 万个网页的耗时过长（正如本章开始时提到的，串行下载需要花费超过 11 天的时间）。

4.2　串行爬虫

下面是串行下载时，之前开发的链接爬虫使用 AlexaCallback 回调的代码。

```
scrape_callback = AlexaCallback()
link_crawler(seed_url=scrape_callback.seed_url,
    cache_callback=MongoCache(),
    scrape_callback=scrape_callback)
```

完整源码可以从 https://bitbucket.org/wswp/code/src/tip/chapter04/sequential_test.py 获取，我们可以在命令行中执行如下命令运行该脚本。

```
$ time python sequential_test.py
...
26m41.141s
```

根据该执行结果估算，串行下载时平均每个 URL 需要花费 1.6 秒。

4.3　多线程爬虫

现在，我们将串行下载网页的爬虫扩展成并行下载。需要注意的是，如果滥用这一功能，多线程爬虫请求内容速度过快，可能会造成服务器过载，或是 IP 地址被封禁。为了避免这一问题，我们的爬虫将会设置一个 delay 标识，用于设定请求同一域名时的最小时间间隔。

作为本章示例的 Alexa 网站列表由于包含了 100 万个不同的域名，因而不会出现上述问题。但是，当你以后爬取同一域名下的不同网页时，就需要注意两次下载之间至少需要 1 秒钟的延时。

4.3.1 线程和进程如何工作

图 4.2 所示为一个包含有多个线程的进程的执行过程。

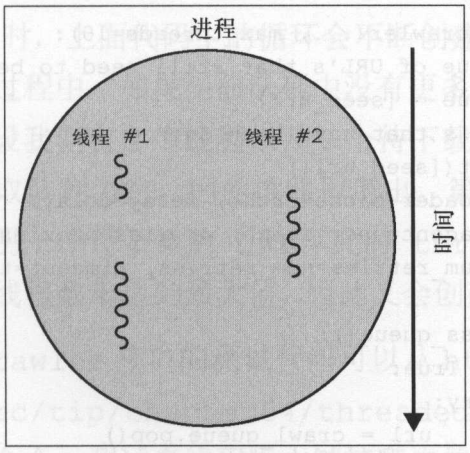


图 4.2

当运行 Python 脚本或其他计算机程序时，就会创建包含有代码和状态的进程。这些进程通过计算机的一个或多个 CPU 来执行。不过，同一时刻每个 CPU 只会执行一个进程，然后在不同进程间快速切换，这样就给人以多个程序同时运行的感觉。同理，在一个进程中，程序的执行也是在不同线程间进行切换的，每个线程执行程序的不同部分。这就意味着当一个线程等待网页下载时，进程可以切换到其他线程执行，避免浪费 CPU 时间。因此，为了充分利用计算机中的所有资源尽可能快地下载数据，我们需要将下载分发到多个进程和线程中。

4.3.2 实现

幸运的是，在 Python 中实现多线程编程相对来说比较简单。我们可以保留与第 1 章开发的链接爬虫类似的队列结构，只是改为在多个线程中启动爬虫循环，以便并行下载这些链接。下面的代码是修改后的链接爬虫起始部分，这里把 `crawl` 循环移到了函数内部。

```
import time
import threading
from downloader import Downloader
SLEEP_TIME = 1

def threaded_crawler(..., max_threads=10):
    # the queue of URL's that still need to be crawled
    crawl_queue = [seed_url]
    # the URL's that have been seen
    seen = set([seed_url])
    D = Downloader(cache=cache, delay=delay,
        user_agent=user_agent, proxies=proxies,
            num_retries=num_retries, timeout=timeout)

    def process_queue():
        while True:
            try:
                url = crawl_queue.pop()
            except IndexError:
                # crawl queue is empty
                break
            else:
                html = D(url)
                ...
```

下面是 threaded_crawler 函数的剩余部分，这里在多个线程中启动了 process_queue，并等待其完成。

```
    threads = []
    while threads or crawl_queue:
        # the crawl is still active
        for thread in threads:
            if not thread.is_alive():
                # remove the stopped threads
                threads.remove(thread)
        while len(threads) < max_threads and crawl_queue:
            # can start some more threads
            thread = threading.Thread(target=process_queue)
            # set daemon so main thread can exit when receives ctrl-c
            thread.setDaemon(True)
            thread.start()
```

```
            threads.append(thread)

        # all threads have been processed
        # sleep temporarily so CPU can focus execution elsewhere
        time.sleep(SLEEP_TIME))
```

当有 URL 可爬取时，上面代码中的循环会不断创建线程，直到达到线程池的最大值。在爬取过程中，如果当前队列中没有更多可以爬取的 URL 时，线程会提前停止。假设我们有 2 个线程以及 2 个待下载的 URL。当第一个线程完成下载时，待爬取队列为空，因此该线程退出。第二个线程稍后也完成了下载，但又发现了另一个待下载的 URL。此时 thread 循环注意到还有 URL 需要下载，并且线程数未达到最大值，因此又会创建一个新的下载线程。

对 threaded_crawler 接口的测试代码可以从 https://bitbucket.org/wswp/code/src/tip/chapter04/threaded_test.py 获取。现在，让我们使用如下命令，测试多线程版本链接爬虫的性能。

```
$ time python threaded_test.py 5
...
4m50.465s
```

由于我们使用了 5 个线程，因此下载速度几乎是串行版本的 5 倍。在 4.4 节中会对多线程性能进行更进一步的分析。

4.3.3 多进程爬虫

为了进一步改善性能，我们对多线程示例再度扩展，使其支持多进程。目前，爬虫队列都是存储在本地内存当中，其他进程都无法处理这一爬虫。为了解决该问题，需要把爬虫队列转移到 MongoDB 当中。单独存储队列，意味着即使是不同服务器上的爬虫也能够协同处理同一个爬虫任务。

请注意，如果想要拥有更加健壮的队列，则需要考虑使用专用的消息传输工具，比如 Celery。不过，为了尽量减少本书中介绍的技术种类，我们在这里选择复用 MongoDB。下面是基于 MongoDB 实现的队列代码。

```
from datetime import datetime, timedelta
from pymongo import MongoClient, errors

class MongoQueue:
    # possible states of a download
    OUTSTANDING, PROCESSING, COMPLETE = range(3)

    def __init__(self, client=None, timeout=300):
        self.client = MongoClient() if client is None else client
        self.db = self.client.cache
        self.timeout = timeout

    def __nonzero__(self):
        """Returns True if there are more jobs to process
        """
        record = self.db.crawl_queue.find_one(
            {'status': {'$ne': self.COMPLETE}}
        )
        return True if record else False

    def push(self, url):
        """Add new URL to queue if does not exist
        """
        try:
            self.db.crawl_queue.insert({'_id': url, 'status':
                self.OUTSTANDING})
        except errors.DuplicateKeyError as e:
            pass # this is already in the queue

    def pop(self):
        """Get an outstanding URL from the queue and set its
        status to processing. If the queue is empty a KeyError
            exception is raised.
        """
        record = self.db.crawl_queue.find_and_modify(
            query={'status': self.OUTSTANDING},
            update={'$set': {'status': self.PROCESSING,
                'timestamp': datetime.now()}}
        )
```

```
        if record:
            return record['_id']
        else:
            self.repair()
            raise KeyError()

    def complete(self, url):
        self.db.crawl_queue.update({'_id': url}, {'$set':
            {'status': self.COMPLETE}})

    def repair(self):
        """Release stalled jobs
        """
        record = self.db.crawl_queue.find_and_modify(
            query={
                'timestamp': {'$lt': datetime.now() -
                    timedelta(seconds=self.timeout)},
                'status': {'$ne': self.COMPLETE}
            },
            update={'$set': {'status': self.OUTSTANDING}}
        )
        if record:
            print 'Released:', record['_id']
```

上面代码中的队列定义了 3 种状态：OUTSTANDING、PROCESSING 和 COMPLETE。当添加一个新 URL 时，其状态为 OUTSTANDING；当 URL 从队列中取出准备下载时，其状态为 PROCESSING；当下载结束后，其状态为 COMPLETE。该实现中，大部分代码都在关注从队列中取出的 URL 无法正常完成时的处理，比如处理 URL 的进程被终止的情况。为了避免丢失这些 URL 的结果，该类使用了一个 timeout 参数，其默认值为 300 秒。在 repair()方法中，如果某个 URL 的处理时间超过了这个 timeout 值，我们就认定处理过程出现了错误，URL 的状态将被重新设为 OUTSTANDING，以便再次处理。

为了支持这个新的队列类型，还需要对多线程爬虫的代码进行少量修改，下面的代码中已经对修改部分进行了加粗处理。

```
def threaded_crawler(...):
    ...
    # the queue of URL's that still need to be crawled
    crawl_queue = MongoQueue()
    crawl_queue.push(seed_url)

    def process_queue():
        while True:
            # keep track that are processing url
            try:
                url = crawl_queue.pop()
            except KeyError:
                # currently no urls to process
                break
            else:
                ...
                crawl_queue.complete(url)
```

第一个改动是将 Python 内建队列替换成基于 MongoDB 的新队列，这里将其命名为 MongoQueue。由于该队列会在内部实现中处理重复 URL 的问题，因此不再需要 seen 变量。最后，在 URL 处理结束后调用 complete() 方法，用于记录该 URL 已经被成功解析。

更新后的多线程爬虫还可以启动多个进程，如下面的代码所示。

```
import multiprocessing

def process_link_crawler(args, **kwargs):
    num_cpus = multiprocessing.cpu_count()
    print 'Starting {} processes'.format(num_cpus)
    processes = []
    for i in range(num_cpus):
        p = multiprocessing.Process(target=threaded_crawler,
            args=[args], kwargs=kwargs)
        p.start()
        processes.append(p)
    # wait for processes to complete
    for p in processes:
        p.join()
```

这段代码的结构看起来十分熟悉，因为多进程模块和之前使用的多线程模块接口相似。这段代码中首先获取可用 CPU 的个数，在每个新进程中启动多线程爬虫，然后等待所有进程完成执行。

现在，让我们使用如下命令，测试多进程版本链接爬虫的性能。测试 process_link_crawler 的接口和之前测试多线程爬虫时一样，可以从 https://bitbucket.org/wswp/code/src/tip/chapter04/process_test.py 获取。

```
$ time python process_test.py 5
Starting 2 processes
...
2m5.405s
```

通过脚本检测，测试服务器包含 2 个 CPU，运行时间大约是之前使用单一进程执行多线程爬虫时的一半。在下一节中，我们将进一步研究这三种方式的相对性能。

4.4 性能

为了进一步了解增加线程和进程的数量会如何影响下载时间，我们对爬取 1000 个网页时的结果进行了对比，如表 4.1 所示。

表 4.1

脚本	线程数	进程数	时间	相对串行的时间比
串行	1	1	28 分 59.966 秒	1
多线程	5	1	7 分 11.634 秒	4.03
多线程	10	1	3 分 50.455 秒	7.55
多线程	20	1	2 分 45.412 秒	10.52
多进程	5	2	4 分 2.624 秒	7.17
多进程	10	2	2 分 1.445 秒	14.33
多进程	20	2	1 分 47.663 秒	16.16

表格的最后一列给出的是相对于串行下载的时间比。可以看出，性能的增长与线程和进程的数量并不是成线性比例的，而是趋于对数。比如，使用 1 个进程和 5 个线程时，性能大约为串行时的 4 倍，而使用 20 个线程时性能只达到了串行下载时的 10 倍。虽然新增的线程能够加快下载速度，但是起到的效果相比于之前添加的线程会越来越小。其实这是可以预见到的现象，因为此时进程需要在更多线程之间进行切换，专门用于每一个线程的时间就会变少。此外，下载的带宽是有限的，最终添加新线程将无法带来更快的下载速度。因此，要想获得更好的性能，就需要在多台服务器上分布式部署爬虫，并且所有服务器都要指向同一个 MongoDB 队列实例。

4.5　本章小结

本章中，我们介绍了串行下载存在瓶颈的原因，然后给出了通过多线程和多进程高效下载大量网页的方法。

下一章中，我们将介绍如何抓取使用 JavaScript 动态加载内容的网页。

第 5 章
动态内容

根据联合国全球网站可访问性审计报告，73%的主流网站都在其重要功能中依赖 JavaScript（参考 `http://www.un.org/esa/socdev/enable/documents/execsumnomensa.doc`）。和单页面应用的简单表单事件不同，使用 JavaScript 时，不再是加载后立即下载所有页面内容。这样就会造成许多网页在浏览器中展示的内容不会出现在 HTML 源代码中，本书前面介绍的抓取技术也就无法正常运转了。对于这种依赖 JavaScript 的动态网站，本章将会介绍两种抓取其数据的方法，分别是：

- JavaScript 逆向工程；
- 渲染 JavaScript。

5.1 动态网页示例

让我们来看一个动态网页的例子。示例网站有一个搜索表单，可以通过 `http://example.webscraping.com/search` 进行访问，该页面用于查询国家。比如说，我们想要查找所有起始字母为 A 的国家，其搜索结果页面如图 5.1 所示。

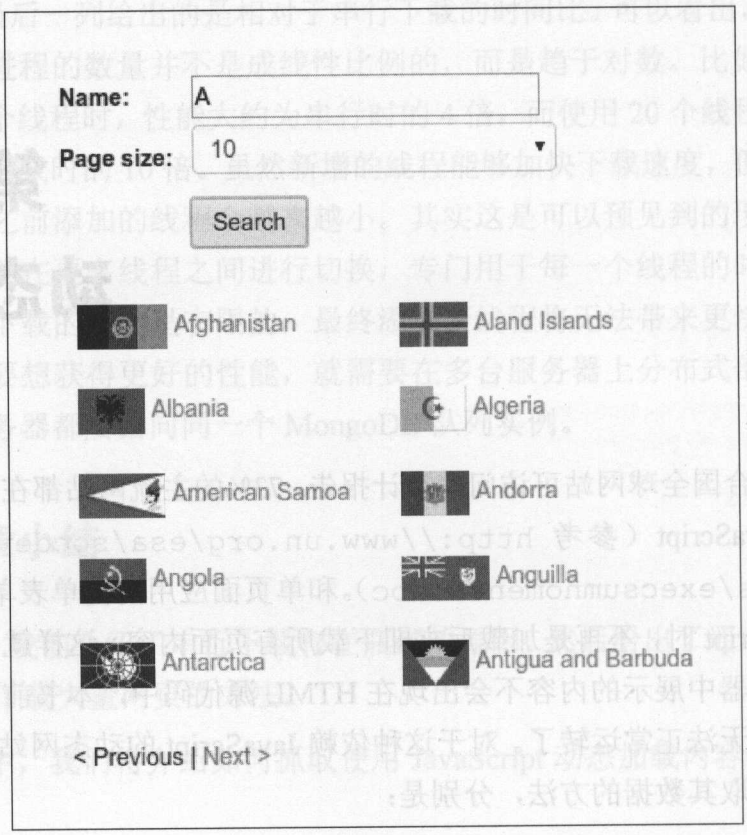


图 5.1

如果我们右键单击结果部分，使用 Firebug 查看元素（参见第 2 章），可以发现结果被存储在 ID 为“result”的 div 元素中，如图 5.2 所示。

让我们尝试使用 lxml 模块抽取这些结果，这里用到的知识在第 2 章和第 3 章的 Downloader 类中都已经介绍过了。

```
>>> import lxml.html
>>> from downloader import Downloader
>>> D = Downloader()
>>> html = D('http://example.webscraping.com/search')
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('div#results a')
 []
```

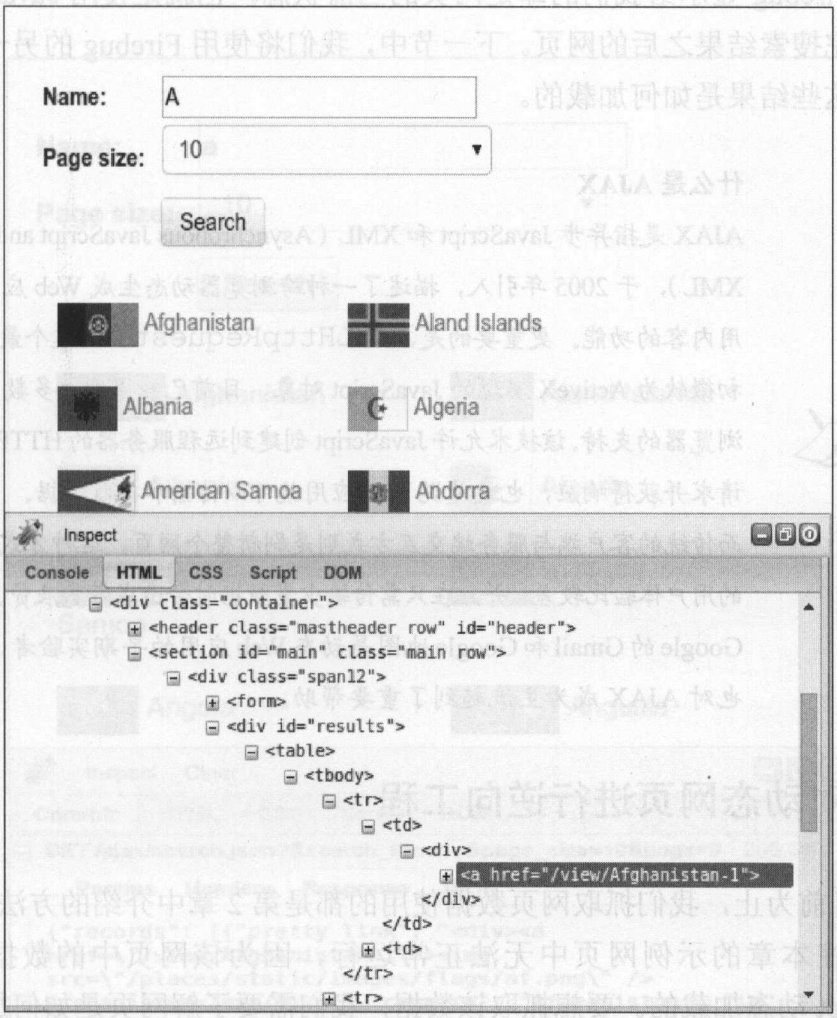


图 5.2

这个示例爬虫在抽取结果时失败了。检查网页源代码可以帮助我们了解抽取操作为什么会失败。在源代码中，可以发现我们准备抓取的 div 元素实际上是空的，如下所示。

```
<div id="results">
</div>
```

而 Firebug 显示给我们的却是网页的当前状态，也就是使用 JavaScript 动态加载完搜索结果之后的网页。下一节中，我们将使用 Firebug 的另一个功能来了解这些结果是如何加载的。

什么是 AJAX

AJAX 是指异步 JavaScript 和 XML（Asynchronous JavaScript and XML），于 2005 年引入，描述了一种跨浏览器动态生成 Web 应用内容的功能。更重要的是，XMLHttpRequest——这个最初微软为 ActiveX 实现的 JavaScript 对象，目前已经得到大多数浏览器的支持。该技术允许 JavaScript 创建到远程服务器的 HTTP 请求并获得响应，也就是说 Web 应用就可以传输和接收数据。而传统的客户端与服务端交互方式则是刷新整个网页，这种方式的用户体验比较差，并且在只需传输少量数据时会造成带宽浪费。Google 的 Gmail 和 Google 地图是动态 Web 应用的早期实验者，也对 AJAX 成为主流起到了重要帮助。

5.2　对动态网页进行逆向工程

到目前为止，我们抓取网页数据使用的都是第 2 章中介绍的方法。但是，该方法在本章的示例网页中无法正常运行，因为该网页中的数据是使用 JavaScript 动态加载的。要想抓取该数据，我们需要了解网页是如何加载该数据的，该过程也被称为逆向工程。继续上一节的例子，在 Firebug 中单击 **Console** 选项卡，然后执行一次搜索，我们将会看到产生了一个 AJAX 请求，如图 5.3 所示。

这个 AJAX 数据不仅可以在搜索网页时访问到，也可以直接下载，如下面的代码所示。

```
>>> html =
    D('http://example.webscraping.com/ajax/
```

```
search.json?page=0&page_size=10&search_term=a')
```

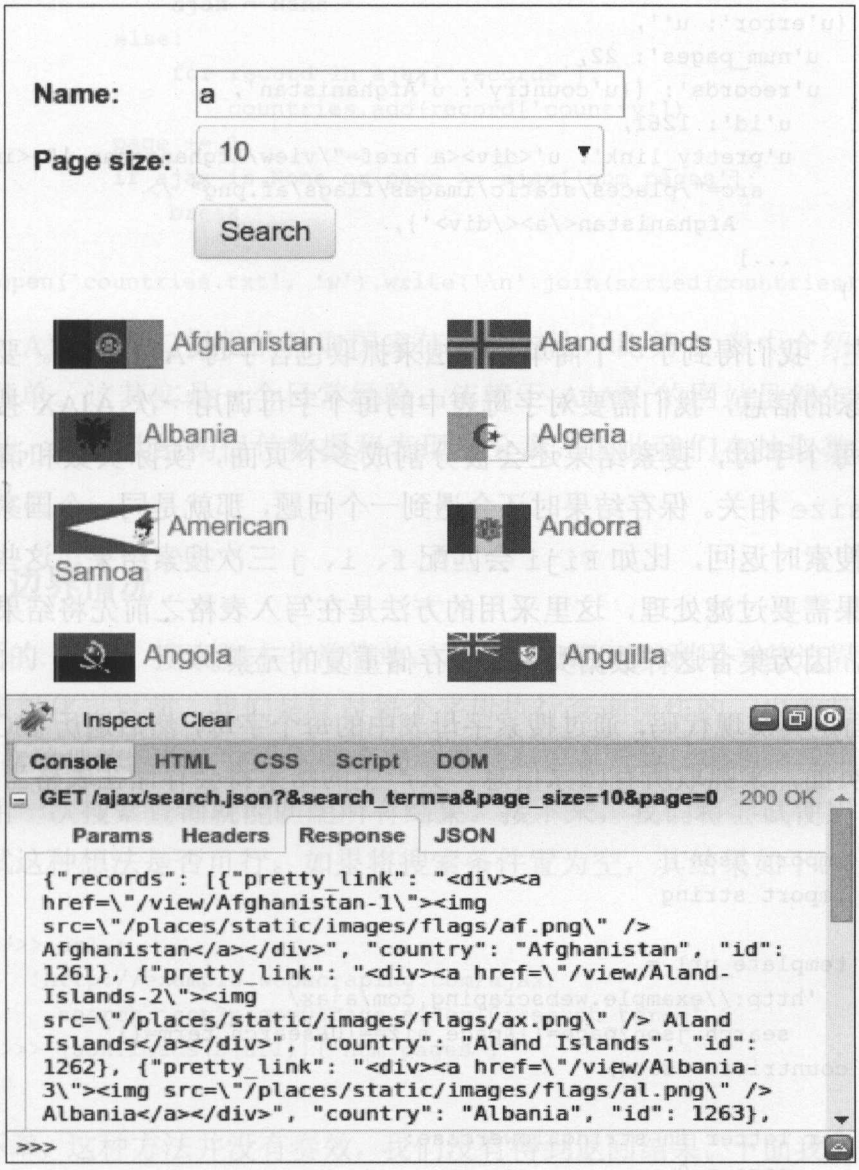


图 5.3

AJAX 响应返回的数据是 JSON 格式的，因此我们可以使用 Python 的 `json` 模块将其解析成一个字典，其代码如下所示。

```
>>> import json
>>> json.loads(html)
{u'error': u'',
 u'num_pages': 22,
 u'records': [{u'country': u'Afghanistan',
   u'id': 1261,
   u'pretty_link': u'<div><a href="/view/Afghanistan-1"><img
     src="/places/static/images/flags/af.png" />
       Afghanistan</a></div>'},
   ...]
}
```

现在，我们得到了一个简单的方法来抓取包含字母 A 的国家。要想获取所有国家的信息，我们需要对字母表中的每个字母调用一次 AJAX 搜索。而且对于每个字母，搜索结果还会被分割成多个页面，实际页数和请求时的 page_size 相关。保存结果时还会遇到一个问题，那就是同一个国家可能会在多次搜索时返回，比如 Fiji 会匹配 f、i、j 三次搜索结果。这些重复的搜索结果需要过滤处理，这里采用的方法是在写入表格之前先将结果存储到集合中，因为集合这种数据类型不会存储重复的元素。

下面是其实现代码，通过搜索字母表中的每个字母，然后遍历 JSON 响应的结果页面，来抓取所有国家信息。其产生的结果将会存储在表格当中。

```
import json
import string

template_url =
  'http://example.webscraping.com/ajax/
    search.json?page={}&page_size=10&search_term={}'
countries = set()
for letter in string.lowercase:
    page = 0
    while True:
        html = D(template_url.format(page, letter))
        try:
            ajax = json.loads(html)
```

```
            except ValueError as e:
                print e
                ajax = None
            else:
                for record in ajax['records']:
                    countries.add(record['country'])
            page += 1
            if ajax is None or page >= ajax['num_pages']:
                break

    open('countries.txt', 'w').write('\n'.join(sorted(countries)))
```

这个 AJAX 接口提供的抽取国家信息的方法，比第 2 章中介绍的抓取方法更简单。这其实是一个日常经验：依赖于 AJAX 的网站虽然乍看起来更加复杂，但是其结构促使数据和表现层分离，因此我们在抽取数据时会更加容易。

5.2.1 边界情况

前面的 AJAX 搜索脚本非常简单，不过我们还可以利用一些边界情况使其进一步简化。目前，我们是针对每个字母执行查询操作的，也就是说我们需要 26 次单独的查询，并且这些查询结果又有很多重复。理想情况下，我们可以使用一次搜索查询就能匹配所有结果。接下来，我们将尝试使用不同字符来测试这种想法是否可行。如果将搜索条件置为空，其结果如下。

```
>>> url =
'http://example.webscraping.com/ajax/
  search.json?page=0&page_size=10&search_term='
>>> json.loads(D(url))['num_pages']
0
```

很不幸，这种方法并没有奏效，我们没有得到返回结果。下面我们再来尝试'*'是否能够匹配所有结果。

```
>>> json.loads(D(url + '*'))['num_pages']
0
```

依然没有奏效。现在我们再来尝试下'.'，这是正则表达式里用于匹配所有字符的元字符。

```
>>> json.loads(D(url + '.'))['num_pages']
26
```

这次尝试成功了，看来服务端是通过正则表达式进行匹配的。因此，现在可以把依次搜索每个字符替换成只对点号搜索一次了。

此外，你可能已经注意到在 AJAX 的 URL 中有一个用于设定每个页面显示国家数量的参数。搜索界面中包含 4、10、20 这几种选项，其中默认值为 10。因此，提高每个页面的显示数量到最大值，可以使下载次数减半。

```
>>> url =
  'http://example.webscraping.com/ajax/
    search.json?page=0&page_size=20&search_term=.'
>>> json.loads(D(url))['num_pages']
13
```

那么，要是使用比网页界面选择框支持的每页国家数更高的数值又会怎样呢？

```
>>> url =
  'http://example.webscraping.com/ajax/
    search.json?page=0&page_size=1000&search_term=.'
>>> json.loads(D(url))['num_pages']
1
```

显然，服务端并没有检查该参数是否与界面允许的选项值相匹配，而是直接在一个页面中返回了所有结果。许多 Web 应用不会在 AJAX 后端检查这一参数，因为它们认为请求只会来自 Web 界面。

现在，我们手工修改了这个 URL，使其能够在一次请求中下载得到所有国家的数据。进一步简化之后，抓取所有国家信息的实现代码如下。

```
        FIELDS = ('area', 'population', 'iso', 'country', 'capital', 'continent',
'tld',  'currency_code',  'currency_name',  'phone',  'postal_code_format',
'postal_code_regex', 'languages', 'neighbours')
```

```
writer = csv.writer(open('countries.csv', 'w'))
writer.writerow(FIELDS)
html = D('http://example.webscraping.com/ajax/
    search.json?page=0&page_size=1000&search_term=.')
ajax = json.loads(html)
for record in ajax['records']:
    row = [record[field] for field in FIELDS]
    writer.writerow(row)
```

5.3 渲染动态网页

对于搜索网页这个例子，我们可以很容易地对其运行过程实施逆向工程。但是，一些网站非常复杂，即使使用类似 Firebug 这样的工具也很难理解。比如，一个网站使用 **Google Web Toolkit**（**GWT**）开发，那么它产生的 JavaScript 代码是机器生成的压缩版。生成的 JavaScript 代码虽然可以使用类似 `JSbeautifier` 的工具进行还原，但是其产生的结果过于冗长，而且原始的变量名也已经丢失，这就会造成其结果难以处理。尽管经过足够的努力，任何网站都可以被逆向工程，但我们可以使用浏览器渲染引擎避免这些工作，这种渲染引擎是浏览器在显示网页时解析 HTML、应用 CSS 样式并执行 JavaScript 语句的部分。在本节中，我们将使用 WebKit 渲染引擎，通过 Qt 框架可以获得该引擎的一个便捷 Python 接口。

什么是 WebKit?

WebKit 的代码源于 1998 年的 KHTML 项目，当时它是 Konqueror 浏览器的渲染引擎。2001 年，苹果公司将该代码衍生为 WebKit，并应用于 Safari 浏览器。Google 在 Chrome 27 之前的版本也使用了 WebKit 内核，直到 2013 年转向利用 WebKit 开发的 Blink 内核。Opera 在 2003 年到 2012 年使用的是其内部的 Presto 渲染引擎，之后切换到 WebKit，但是不久又跟随 Chrome 转向 Blink。其他主流渲染引擎还包括 IE 使用的 Trident 和 Firefox 的 Gecko。

5.3.1　PyQt 还是 PySide

Qt 框架有两种可以使用的 Python 库，分别是 PyQt 和 PySide。PyQt 最初于 1998 年发布，但在用于商业项目时需要购买许可。由于该原因，开发 Qt 的公司（原先是诺基亚，现在是 Digia）后来在 2009 年开发了另一个 Python 库 PySide，并且使用了更加宽松的 LGPL 许可。

虽然这两个库有少许区别，但是本章中的例子在两个库中都能够正常工作。下面的代码片段用于导入已安装的任何一种 Qt 库。

```
try:
    from PySide.QtGui import *
    from PySide.QtCore import *
    from PySide.QtWebKit import *
except ImportError:
    from PyQt4.QtGui import *
    from PyQt4.QtCore import *
    from PyQt4.QtWebKit import *
```

在这段代码中，如果 PySide 不可用，则会抛出 ImportError 异常，然后导入 PyQt 模块。如果 PyQt 模块也不可用，则会抛出另一个 ImportError 异常，然后退出脚本。

下载和安装这两种 Python 库的说明可以分别参考 http://qt-project.org/wiki/Setting_up_PySide 和 http://pyqt.sourceforge.net/Docs/PyQt4/installation.html。

5.3.2　执行 JavaScript

为了确认 WebKit 能够执行 JavaScript，我们可以使用位于 http://example.webscraping.com/dynamic 上的这个简单示例。

该网页只是使用 JavaScript 在 div 元素中写入了 Hello World。下面是其源代码。

```
<html>
    <body>
        <div id="result"></div>
        <script>
        document.getElementById("result").innerText = 'Hello World';
        </script>
    </body>
</html>
```

使用传统方法下载原始 HTML 并解析结果时，得到的 div 元素为空值，如下所示。

```
>>> url = 'http://example.webscraping.com/dynamic'
>>> html = D(url)
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('#result')[0].text_content()
''
```

下面是使用 WebKit 的初始版本代码，当然还需事先导入上一节提到的 PyQt 或 PySide 模块。

```
>>> app = QApplication([])
>>> webview = QWebView()
>>> loop = QEventLoop()
>>> webview.loadFinished.connect(loop.quit)
>>> webview.load(QUrl(url))
>>> loop.exec_()
>>> html = webview.page().mainFrame().toHtml()
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('#result')[0].text_content()
'Hello World'
```

因为这里有很多新知识，所以下面我们会逐行分析这段代码。

- 第一行初始化了 QApplication 对象，在其他 Qt 对象完成初始化之前，Qt 框架需要先创建该对象。

- 接下来，创建 QWebView 对象，该对象是 Web 文档的容器。
- 创建 QEventLoop 对象，该对象用于创建本地事件循环。
- QWebView 对象的 loadFinished 回调连接了 QEventLoop 的 quit 方法，从而可以在网页加载完成之后停止事件循环。然后，将要加载的 URL 传给 QWebView。PyQt 需要将该 URL 字符串封装在 QUrl 对象当中，而对于 PySide 来说则是可选项。
- 由于 QWebView 的加载方法是异步的，因此执行过程会在网页加载时立即传入下一行。但我们又希望等待网页加载完成，因此需要在事件循环启动时调用 loop.exec_()。
- 网页加载完成后，事件循环退出，执行过程移到下一行，对加载得到的网页所产生的 HTML 进行数据抽取。
- 从最后一行可以看出，我们成功执行了 JavaScript，div 元素果然抽取出了 Hello World。

这里使用的类和方法在 C++的 Qt 框架网站中都有详细的文档，其网址为 http://qt-project.org/doc/qt-4.8/。虽然 PyQt 和 PySide 都有其自身的文档，但是原始 C++版本的描述和格式更加详尽，一般的 Python 开发者可以用它替代。

5.3.3　使用 WebKit 与网站交互

我们用于测试的搜索网页需要用户修改后提交搜索表单，然后单击页面链接。而前面介绍的浏览器渲染引擎只能执行 JavaScript，然后访问生成的 HTML。要想抓取搜索页面，我们还需要对浏览器渲染引擎进行扩展，使其支持交互功能。幸运的是，Qt 包含了一个非常棒的 API，可以选择和操纵 HTML 元素，使交互操作变得简单。

对于之前的 AJAX 搜索示例，下面给出另一个实现版本。该版本已经将搜索条件设为'.'，每页显示数量设为'1000'，这样只需一次请求就能获取

到全部结果。

```
app = QApplication([])
webview = QWebView()
loop = QEventLoop()
webview.loadFinished.connect(loop.quit)
webview.load(QUrl('http://example.webscraping.com/search'))
loop.exec_()
webview.show()
frame = webview.page().mainFrame()
frame.findFirstElement('#search_term').
       setAttribute('value', '.')
frame.findFirstElement('#page_size option:checked').
       setPlainText('1000')
frame.findFirstElement('#search').
       evaluateJavaScript('this.click()')
app.exec_()
```

最开始几行和之前的 Hello World 示例一样，初始化了一些用于渲染网页的 Qt 对象。之后，调用 QWebView GUI 的 show() 方法来显示渲染窗口，这可以方便调试。然后，创建了一个指代框架的变量，可以让后面几行代码更短。QWebFrame 类有很多与网页交互的有用方法。接下来的两行使用 CSS 模式在框架中定位元素，然后设置搜索参数。而后表单使用 evaluateJavaScript() 方法进行提交，模拟点击事件。该方法非常实用，因为它允许我们插入任何想要的 JavaScript 代码，包括直接调用网页中定义的 JavaScript 方法。最后一行进入应用的事件循环，此时我们可以对表单操作进行复查。如果没有使用该方法，脚本将会直接结束。

图 5.4 所示为脚本运行时的显示界面。

1. 等待结果

实现 WebKit 爬虫的最后一部分是抓取搜索结果，而这又是最难的一部分，因为我们难以预估完成 AJAX 事件以及准备好国家数据的时间。有三种方法可以处理这一问题，分别是：

- 等待一定时间，期望 AJAX 事件能够在此时刻之前完成；

- 重写 Qt 的网络管理器，跟踪 URL 请求的完成时间；
- 轮询网页，等待特定内容出现。

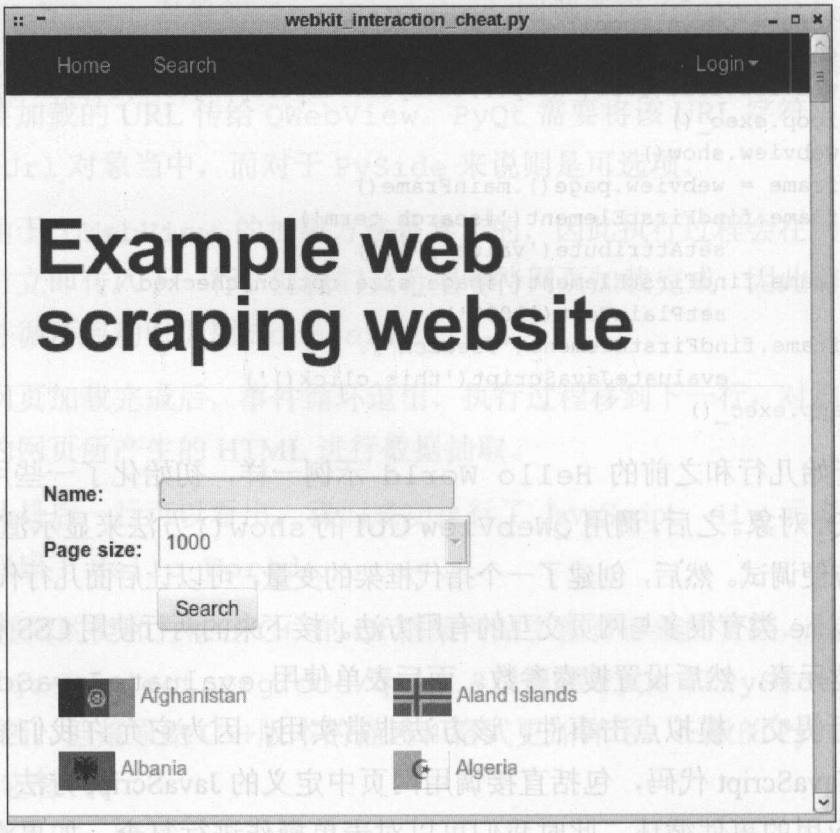


图 5.4

第一种方案最容易实现，不过效率也最低，因为一旦设置了安全的超时时间，就会使大多数请求浪费大量不必要的时间。而且，当网络速度比平常慢时，固定的超时时间会出现请求失败的情况。第二种方案虽然更加高效，但是如果延时出现在客户端而不是服务端时，则无法使用。比如，已经完成下载，但是需要再单击一个按钮才会显示内容这种情况，延时就出现在客户端。第三种方案尽管存在一个小缺点，即会在检查内容是否加载完成时浪费 CPU 周期，但是该方案更加可靠且易于实现。下面是使用第三种方案的实现代码。

```
>>> elements = None
>>> while not elements:
...     app.processEvents()
...     elements = frame.findAllElements('#results a')
...
>>> countries = [e.toPlainText().strip() for e in elements]
>>> print countries
 [u'Afghanistan', u'Aland Islands', ... , u'Zambia', u'Zimbabwe']
```

如上实现中，代码不断循环，直到国家链接出现在 results 这个 div 元素中。每次循环，都会调用 app.processEvents()，用于给 Qt 事件循环执行任务的时间，比如响应点击事件和更新 GUI。

2. 渲染类

为了提升这些功能后续的易用性，下面会把使用到的方法封装到一个类中，其源代码可以从 https://bitbucket.org/wswp/code/src/tip/chapter05/browser_render.py 获取。

```
import time

class BrowserRender(QWebView):
    def __init__(self, show=True):
        self.app = QApplication(sys.argv)
        QWebView.__init__(self)
        if show:
            self.show() # show the browser

    def download(self, url, timeout=60):
        """Wait for download to complete and return result"""
        loop = QEventLoop()
        timer = QTimer()
        timer.setSingleShot(True)
        timer.timeout.connect(loop.quit)
        self.loadFinished.connect(loop.quit)
        self.load(QUrl(url))
        timer.start(timeout * 1000)
        loop.exec_() # delay here until download finished
```

```
            if timer.isActive():
                # downloaded successfully
                timer.stop()
                return self.html()
            else:
                # timed out
                print 'Request timed out: ' + url

    def html(self):
        """Shortcut to return the current HTML"""
        return self.page().mainFrame().toHtml()

    def find(self, pattern):
        """Find all elements that match the pattern"""
        return self.page().mainFrame().findAllElements(pattern)

    def attr(self, pattern, name, value):
        """Set attribute for matching elements"""
        for e in self.find(pattern):
            e.setAttribute(name, value)

    def text(self, pattern, value):
        """Set attribute for matching elements"""
        for e in self.find(pattern):
            e.setPlainText(value)

    def click(self, pattern):
        """Click matching elements"""
        for e in self.find(pattern):
            e.evaluateJavaScript("this.click()")

    def wait_load(self, pattern, timeout=60):
        """Wait until pattern is found and return matches"""
        deadline = time.time() + timeout
        while time.time() < deadline:
            self.app.processEvents()
            matches = self.find(pattern)
            if matches:
                return matches
        print 'Wait load timed out'
```

你可能已经注意到，在 download()和 wait_load()方法中我们增加了一些代码用于处理定时器。定时器用于跟踪等待时间，并在截止时间到达时取消事件循环。否则，当出现网络问题时，事件循环就会无休止地运行下去。

下面是使用这个新实现的类抓取搜索页面的代码。

```
>>> br = BrowserRender()
>>> br.download('http://example.webscraping.com/search')
>>> br.attr('#search_term', 'value', '.')
>>> br.text('#page_size option:checked', '1000')
>>> br.click('#search')
>>> elements = br.wait_load('#results a')
>>> countries = [e.toPlainText().strip() for e in elements]
>>> print countries
[u'Afghanistan', u'Aland Islands', ..., u'Zambia', u'Zimbabwe']
```

5.3.4 Selenium

使用前面例子中的 WebKit 库，我们可以自定义浏览器渲染引擎，这样就能完全控制想要执行的行为。如果不需要这么高的灵活性，那么还有一个不错的替代品 Selenium 可以选择，它提供了使浏览器自动化的 API 接口。Selenium 可以通过如下命令使用 pip 安装。

```
pip install selenium
```

为了演示 Selenium 是如何运行的，我们会把之前的搜索示例重写成 Selenium 的版本。首先，创建一个到浏览器的连接。

```
>>> from selenium import webdriver
>>> driver = webdriver.Firefox()
```

当运行该命令时，会弹出一个空的浏览器窗口，如图 5.5 所示。

该功能非常方便，因为在执行每条命令时，都可以通过浏览器窗口来检查

Selenium 是否依照预期运行。尽管这里我们使用的浏览器是 Firefox，不过 Selenium 也提供了连接其他常见浏览器的接口，比如 Chrome 和 IE。需要注意的是，我们只能使用系统中已安装浏览器的 Selenium 接口。

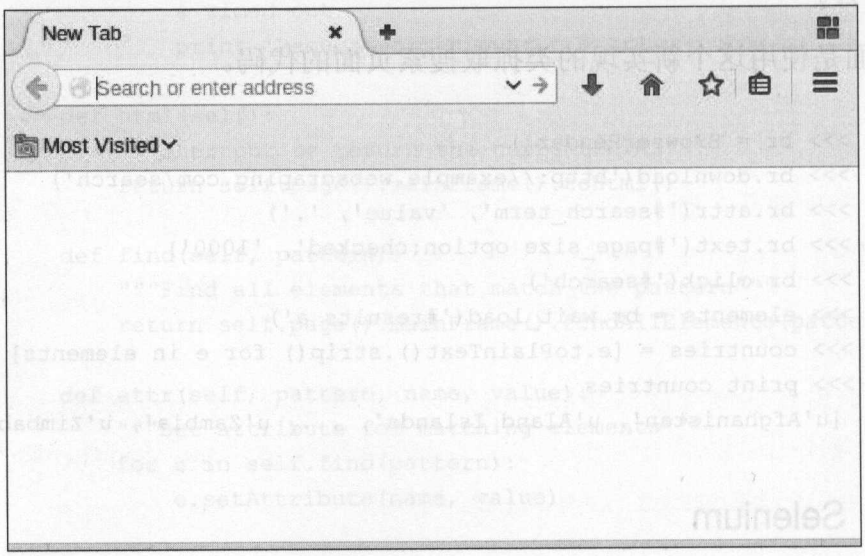


图 5.5

如果想在选定的浏览器中加载网页，可以调用 `get()` 方法：

```
>>> driver.get('http://example.webscraping.com/search')
```

然后，设置需要选取的元素，这里使用的是搜索文本框的 ID。此外，Selenium 也支持使用 CSS 选择器或 XPath 来选取元素。当找到搜索文本框之后，我们可以通过 `send_keys()` 方法输入内容，模拟键盘输入。

```
>>> driver.find_element_by_id('search_term').send_keys('.')
```

为了让所有结果可以在一次搜索后全部返回，我们希望把每页显示的数量设置为 1000。但是，由于 Selenium 的设计初衷是与浏览器交互，而不是修改网页内容，因此这种想法并不容易实现。要想绕过这一限制，我们可以使用 JavaScript 语句直接设置选项框的内容。

```
>>> js = "document.getElementById('page_size').options[1].text ='1000'"
>>> driver.execute_script(js);
```

此时表单内容已经输入完毕，下面就可以单击搜索按钮执行搜索了。

```
>>> driver.find_element_by_id('search').click()
```

现在，我们需要等待 AJAX 请求完成之后才能加载结果，在之前讲解的 WebKit 实现中这里是最难的一部分脚本。幸运的是，Selenium 为该问题提供了一个简单的解决方法，那就是可以通过 implicitly_wait() 方法设置超时时间。

```
>>> driver.implicitly_wait(30)
```

此处，我们设置了 30 秒的延时。如果我们要查找的元素没有出现，Selenium 至多等待 30 秒，然后就会抛出异常。要想选取国家链接，我们依然可以使用 WebKit 示例中用过的那个 CSS 选择器。

```
>>> links = driver.find_elements_by_css_selector('#results a')
```

然后，抽取每个链接的文本，并创建一个国家列表。

```
>>> countries = [link.text for link in links]
>>> print countries
 [u'Afghanistan', u'Aland Islands', ..., u'Zambia', u'Zimbabwe']
```

最后，调用 close() 方法关闭浏览器。

```
>>> driver.close()
```

本示例的源代码可以从 https://bitbucket.org/wswp/code/src/tip/chapter05/selenium_search.py 获取。如果想进一步了解 Selenium 这个 Python 库，可以通过 https://selenium-python.readthedocs.org/ 获取其文档。

5.4　本章小结

本章介绍了两种抓取动态网页数据的方法。第一种方法是借助 Firebug Lite 对动态网页进行逆向工程，第二种方法是使用浏览器渲染引擎为我们触发 JavaScript 事件。我们首先使用 WebKit 创建自定义浏览器，然后使用更高级的 Selenium 框架重新实现该爬虫。

浏览器渲染引擎能够为我们节省了解网站后端工作原理的时间，但是该方法也有其劣势。渲染网页增加了开销，使其比单纯下载 HTML 更慢。另外，使用浏览器渲染引擎的方法通常需要轮询网页来检查是否已经得到事件生成的 HTML，这种方式非常脆弱，在网络较慢时会经常会失败。我一般将浏览器渲染引擎作为短期解决方案，此时长期的性能和可靠性并不算重要；而作为长期解决方案，我会尽最大努力对网站进行逆向工程。

在下一章中，我们将介绍如何与表单进行交互，以及使用 cookie 登录网站并编辑内容。

第 6 章
表单交互

在前面几章中，我们下载的静态网页总是返回相同的内容。而在本章中，我们将与网页进行交互，根据用户输入返回对应的内容。本章将包含如下几个主题：

- 发送 POST 请求提交表单；
- 使用 cookie 登录网站；
- 用于简化表单提交的高级模块 Mechanize。

表单方法

HTML 定义了两种向服务器提交数据的方法，分别是 GET 和 POST。使用 GET 方法时，会将类似?name1=value1&name2=value2 的数据添加到 URL 中，这串数据被称为“查询字符串”。由于浏览器存在 URL 长度限制，因此这种方法只适用于少量数据的场景。另外，这种方法应当用于从服务器端获取数据，而不是修改数据，不过开发者有时会忽视这一规定。而在使用 POST 请求时，数据在请求体中发送，与 URL 保持分离。敏感数据只应使用 POST 请求进行发送，以避免将数据暴露在 URL 中。POST 数据在请求体中如何表示需要依赖于所使用的编码类型。

服务器端还支持其他 HTTP 方法，比如 PUT 和 DELETE 方法，不过这些方法在表单中均不支持。

想要和表单进行交互，就需要拥有可以登录网站的用户账号。现在我们需要手工注册账号，其网址为 http://example.webscraping.com/user/register。本章目前还无法自动化注册表单，不过在下一章我们会介绍处理验证码问题的方法，从而实现自动化表单注册。

6.1　登录表单

我们最先要实施自动化提交的是**登录**表单，其网址为 http://example.webscraping.com/user/login。要想理解该表单，我们需要用到 Firebug Lite。如果使用完整版的 Firebug 或者 Chrome 的开发者工具，我们只需要提交表单就可以在网络选项卡中检查传输的数据。但是，Lite 版本限制我们只能查看结构，如图 6.1 所示。

```
<form action="#" enctype="application/x-www-form-urlencoded" method="post">
  ⊟ <table>
      ⊟ <tbody>
          ⊟ <tr id="auth_user_email__row">
              ⊞ <td class="w2p_fl">
              ⊟ <td class="w2p_fw">
                    <input class="string" id="auth_user_email" name="email" type="text" value=""/>
                </td>
                <td class="w2p_fc"/>
            </tr>
          ⊟ <tr id="auth_user_password__row">
              ⊞ <td class="w2p_fl">
              ⊟ <td class="w2p_fw">
                    <input class="password" id="auth_user_password" name="password" type="password" value=""/>
                </td>
                <td class="w2p_fc"/>
            </tr>
          ⊞ <tr id="auth_user_remember__row">
          ⊞ <tr id="submit_record__row">
        </tbody>
    </table>
  ⊟ <div style="display: none;">
        <input name="_next" type="hidden" value="/"/>
        <input name="_formkey" type="hidden" value="fd155d33-0b81-48de-86f8-6a2b62bcd701"/>
        <input name="_formname" type="hidden" value="login"/>
    </div>
</form>
```

图 6.1

图 6.1 中包括几个重要组成部分，分别是 form 标签的 action、enctype 和 method 属性，以及两个 input 域。action 属性用于设置表单数据提交的地址，本例中为#，也就是和**登录**表单相同的 URL。enctype 属性用于设置

数据提交的编码，本例中为 application/x-www-form-urlencoded。而 method 属性被设为 post，表示通过请求体向服务器端提交表单数据。对于 input 标签，重要的属性是 name，用于设定提交到服务器端时某个域的名称。

表单编码

当表单使用 POST 方法时，表单数据提交到服务器端之前有两种编码类型可供选择。默认编码类型为 application/x-www-form-urlencoded，此时所有非字母数字类型的字符都需要转换为十六进制的 ASCII 值。但是，如果表单中包含大量非字母数字类型的字符时，这种编码类型的效率就会非常低，比如处理二进制文件上传时就存在该问题，此时就需要定义 multipart/form-data 作为编码类型。使用这种编码类型时，不会对输入进行编码，而是使用 MIME 协议将其作为多个部分进行发送，和邮件的传输标准相同。

该标准的官方文档为 http://www.w3.org/TR/html5/forms.html#selectinga-form-submission-encoding。

当普通用户通过浏览器打开该网页时，需要输入邮箱和密码，然后单击**登录**按钮将数据提交到服务端。如果登录成功，则会跳转到主页；否则，会跳转回**登录**页。下面是尝试自动化处理该流程的初始版本代码。

```
>>> import urllib, urllib2
>>> LOGIN_URL = 'http://example.webscraping.com/user/login'
>>> LOGIN_EMAIL = 'example@webscraping.com'
>>> LOGIN_PASSWORD = 'example'
>>> data = {'email': LOGIN_EMAIL, 'password': LOGIN_PASSWORD}
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(LOGIN_URL, encoded_data)
>>> response = urllib2.urlopen(request)
>>> response.geturl()
'http://example.webscraping.com/user/login'
```

上述代码中，我们设置了邮箱和密码域，并将其进行了 urlencode 编码，然后将这些数据提交到服务器端。当执行最后的打印语句时，输出的依然是**登录**页的 URL，也就是说登录失败了。

这是因为**登录**表单十分严格，除邮箱和密码外，还需要提交另外几个域。我们可以从图 6.1 的最下方找到这几个域，不过由于设置为 hidden，所以不会在浏览器中显示出来。为了访问这些隐藏域，下面使用第 2 章中介绍的 lxml 库编写一个函数，提取表单中所有 input 标签的详情。

```
import lxml.html
def parse_form(html):
tree = lxml.html.fromstring(html)
data = {}
for e in tree.cssselect('form input'):
    if e.get('name'):
        data[e.get('name')] = e.get('value')
return data
```

上述代码使用 lxml 的 CSS 选择器遍历表单中所有的 input 标签，然后以字典的形式返回其中的 name 和 value 属性。对**登录**页运行该函数后，得到的结果如下所示。

```
>>> import pprint
>>> html = urllib2.urlopen(LOGIN_URL).read()
>>> form = parse_form(html)
>>> pprint.pprint(form)
{
    '_next': '/',
    '_formkey': '0291ec65-9332-426e-b6a1-d97b3a2b12f8',
    '_formname': 'login',
    'email': '',
    'password': '',
    'remember': 'on'
}
```

其中，_formkey 属性是这里的关键部分，服务器端使用这个唯一的 ID 来避免表单多次提交。每次加载网页时，都会产生不同的 ID，然后服务器端

就可以通过这个给定的 ID 来判断表单是否已经提交过。下面是提交了 _formkey 及其他隐藏域的新版本登录代码。

```
>>> html = urllib2.urlopen(LOGIN_URL).read()
>>> data = parse_form(html)
>>> data['email'] = LOGIN_EMAIL
>>> data['password'] = LOGIN_PASSWORD
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(LOGIN_URL, encoded_data)
>>> response = urllib2.urlopen(request)
>>> response.geturl()
'http://example.webscraping.com/user/login'
```

很遗憾，这个版本依然不能正常工作，运行时还是会打印出登录 URL。这是因为我们缺失了一个重要的组成部分——cookie。当普通用户加载**登录**表单时，_formkey 的值将会保存在 cookie 中，然后该值会与提交的**登录**表单数据中的 _formkey 值进行对比。下面是使用 urllib2.HTTPCookieProcessor 类增加了 cookie 支持之后的代码。

```
>>> import cookielib
>>> cj = cookielib.CookieJar()
>>> opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
>>> html = opener.open(LOGIN_URL).read()
>>> data = parse_form(html)
>>> data['email'] = LOGIN_EMAIL
>>> data['password'] = LOGIN_PASSWORD
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(LOGIN_URL, encoded_data)
>>> response = opener.open(request)
>>> response.geturl()
'http://example.webscraping.com'
```

什么是 cookie?

cookie 是网站在 HTTP 响应头中传输的少量数据，形如：Set-Cookie: session_id=example;。浏览器将会存储这些数据，并在后续对该网站的请求头中包含它们。这样就可以让网站识别和跟踪用户。

这次我们终于成功了！服务器端接受了我们提交的表单值，response 的 URL 是主页。该代码片段以及本章中其他登录示例代码都可以从 https://bitbucket.org/wswp/code/src/tip/chapter06/login.py 获取。

6.1.1　从浏览器加载 cookie

从前面的例子可以看出，如何向服务器提交它所需的登录信息，有时候会很复杂。幸好，对于这种麻烦的网站还有一个变通方法，即先在浏览器中手工执行登录，然后在 Python 脚本中复用之前得到的 cookie，从而实现自动登录。不同浏览器采用不同的格式存储 cookie，这里我们仅以 Firefox 浏览器为例。

Firefox 在 sqlite 数据库中存储 cookie，在 JSON 文件中存储 session，这两种存储方式都可以直接通过 Python 获取。对于登录操作来说，我们只需要获取 session，其存储结构如下所示。

```
{"windows": [...
  "cookies": [
    {"host":"example.webscraping.com",
     "value":"514315085594624:e5e9a0db-5b1f-4c66-a864",
     "path":"/",
     "name":"session_id_places"}
    ...]
]}
```

下面是把 session 解析到 CookieJar 对象的函数代码。

```
def load_ff_sessions(session_filename):
    cj = cookielib.CookieJar()
    if os.path.exists(session_filename):
        json_data = json.loads(open(session_filename, 'rb').read())
        for window in json_data.get('windows', []):
            for cookie in window.get('cookies', []):
                c = cookielib.Cookie(0,
                    cookie.get('name', ''),
                    cookie.get('value', ''), None, False,
                    cookie.get('host', ''),
```

```
                    cookie.get('host', '').startswith('.'),
                    cookie.get('host', '').startswith('.'),
                    cookie.get('path', ''), False, False,
                    str(int(time.time()) + 3600 * 24 * 7),
                    False, None, None, {})
                cj.set_cookie(c)
    else:
        print 'Session filename does not exist:', session_filename
    return cj
```

这里有一个比较麻烦的地方：不同的操作系统中，Firefox 存储 session 文件的位置不同。在 Linux 系统中，其路径如下所示。

```
~/.mozilla/firefox/*.default/sessionstore.js
```

在 OS X 中，其路径如下所示。

```
~/Library/Application Support/Firefox/Profiles/*.default/sessionstore.js
```

而在 Windows Vista 及以上版本系统中，其路径如下所示。

```
%APPDATA%/Roaming/Mozilla/Firefox/Profiles/*.default/sessionstore.js
```

下面是返回 session 文件路径的辅助函数代码。

```
import os, glob
def find_ff_sessions():
    paths = [
        '~/.mozilla/firefox/*.default',
        '~/Library/Application Support/Firefox/Profiles/*.default',
        '%APPDATA%/Roaming/Mozilla/Firefox/Profiles/*.default'
    ]
    for path in paths:
        filename = os.path.join(path, 'sessionstore.js')
        matches = glob.glob(os.path.expanduser(filename))
        if matches:
            return matches[0]
```

需要注意的是，这里使用的 `glob` 模块会返回指定路径中所有匹配的文件。下面是修改后使用浏览器 cookie 登录的代码片段。

```
>>> session_filename = find_ff_sessions()
>>> cj = load_ff_sessions(session_filename)
>>> processor = urllib2.HTTPCookieProcessor(cj)
>>> opener = urllib2.build_opener(processor)
>>> url = 'http://example.webscraping.com'
>>> html = opener.open(url).read()
```

要检查 session 是否加载成功，这次我们无法再依靠登录跳转了。这时我们需要抓取产生的 HTML，检查是否存在登录用户标签。如果得到的结果是 `Login`，则说明 session 没能正确加载。如果出现这种情况，你就需要确认一下 Firefox 中是否已经成功登录示例网站。图 6.2 所示为 Firebug 中显示的用户标签结构。

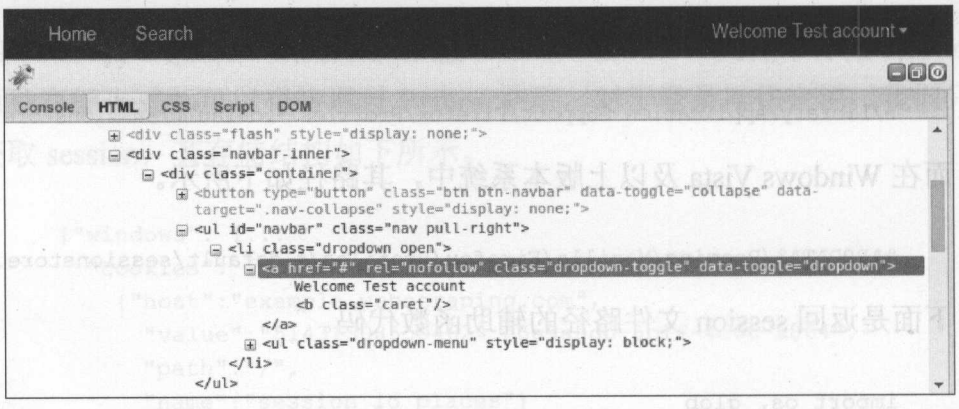


图 6.2

Firebug 中显示该标签位于 ID 为“navbar”的<ul>标签中，我们可以使用第 2 章中介绍的 `lxml` 库抽取其中的信息。

```
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('ul#navbar li a')[0].text_content()
Welcome Test account
```

本节中的代码非常复杂，而且只支持从 Firefox 浏览器中加载 session。如果你想支持其他浏览器的 cookie，可以使用 `browsercookie` 模块。该模块可以通过 `pip install browsercookie` 命令进行安装，其文档地址为 `https://pypi.python.org/pypi/browsercookie`。

6.2 支持内容更新的登录脚本扩展

自动化登录成功运行后，我们还可以继续扩展该脚本，使其与网站进行交互，更新国家数据。本节中使用的代码可以从 `https://bitbucket.org/wswp/code/src/tip/chapter06/edit.py` 获取。如图 6.3 所示，每个国家页面底部均有一个 **Edit** 链接。

图 6.3

在登录情况下，该链接会指向另一个页面，在该页面中所有国家属性都可以进行编辑，如图 6.4 所示。

图 6.4

这里我们编写一个脚本，每次运行时，都会使该国家的人口数量加 1。首先是复用 `parse_form()` 函数，抽取国家人口数量的当前值。

```
>>> import login
>>> COUNTRY_URL = 'http://example.webscraping.com/edit/
    United-Kingdom-239'
>>> opener = login.login_cookies()
>>> country_html = opener.open(COUNTRY_URL).read()
```

```
>>> data = parse_form(country_html)
>>> pprint.pprint(data)
{'_formkey': '4cf0294d-ea71-4cd8-ae2a-43d4ca0d46dd',
 '_formname': 'places/5402840151359488',
 'area': '244820.00',
 'capital': 'London',
 'continent': 'EU',
 'country': 'United Kingdom',
 'currency_code': 'GBP',
 'currency_name': 'Pound',
 'id': '5402840151359488',
 'iso': 'GB',
 'languages': 'en-GB,cy-GB,gd',
 'neighbours': 'IE',
 'phone': '44',
 'population': '62348447',
 'postal_code_format': '@# #@@|@## #@@|@@# #@@|@@## #@@|@#@
   #@@|@@#@ #@@|GIR0AA',
 'postal_code_regex': '^(([A-Z]\\d{2}[A-Z]{2})|([A-Z]\\d{3}
    [A-Z]{2})|([A-Z]{2}\\d{2}[A-Z]{2})|([A-Z]{2}\\d{3}
       [A-Z]{2})|([A-Z]\\d[A-Z]\\d[A-Z]{2})|([A-Z]{2}\\
          d[A-Z]\\d[A-Z]{2})|(GIR0AA))$',
 'tld': '.uk'}
```

然后为人口数量加 1，并将更新提交到服务器端。

```
>>> data['population'] = int(data['population']) + 1
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(COUNTRY_URL, encoded_data)
>>> response = opener.open(request)
```

当我们再次回到国家页时，可以看到人口数量已经增长到 **62,348,448**，如图 6.5 所示。

图 6.5

读者可以对任何字段随意进行修改和测试，因为网站所用的数据库每个小时都会将国家数据恢复为初始值，以保证数据正常。本节中使用的代码可以从 https://bitbucket.org/wswp/code/src/tip/chapter06/edit.py 获取。需要注意的是，严格来说，本例并不算是网络爬虫，而是广义上的网络机器人。不过，这里使用的表单技术可以应用于抓取时的复杂表单交互当中。

6.3　使用 Mechanize 模块实现自动化表单处理

尽管我们的例子现在已经可以正常运行，但是可以发现每个表单都需要大量的工作和测试。我们可以使用 Mechanize 模块减轻这方面的工作，该模块提供了与表单交互的高级接口。Mechanize 可以通过 pip 命令进行安装。

```
pip install mechanize
```

下面是使用 Mechanize 实现前面的人口数量增长示例的代码。

```
>>> import mechanize
>>> br = mechanize.Browser()
>>> br.open(LOGIN_URL)
>>> br.select_form(nr=0)
>>> br['email'] = LOGIN_EMAIL
>>> br['password'] = LOGIN_PASSWORD
>>> response = br.submit()
>>> br.open(COUNTRY_URL)
>>> br.select_form(nr=0)
>>> br['population'] = str(int(br['population']) + 1)
>>> br.submit()
```

这段代码比之前的例子要简单得多，因为我们不再需要管理 cookie，而且访问表单输入框也更加容易。该脚本首先创建一个 Mechanize 浏览器对象，然后定位到登录 URL，选择登录表单。我们可以直接向浏览器对象传递名称

和值，来设置选定表单的输入框内容。调试时，我们可以直接调用 br.form，获取提交之前的表单状态，如下面的代码所示。

```
>>> print br.form
<POST http://example.webscraping.com/user/login# application/
 x-www-form-urlencoded
 <TextControl(email=)>
 <PasswordControl(password=)>
 <CheckboxControl(remember=[on])>
 <SubmitControl(<None>=Login) (readonly)>
 <SubmitButtonControl(<None>=) (readonly)>
 <HiddenControl(_next=/) (readonly)>
 <HiddenControl(_formkey=5fa268b4-0dfd-4e3f-a274-e73c6b7ce584)
   (readonly)>
 <HiddenControl(_formname=login) (readonly)>>
```

设定好登录表单的参数之后，调用 br.submit() 提交选定的登录表单，然后脚本就处于登录状态了。现在，定位到国家的编辑页面，继续使用表单界面增加人口数量。需要注意的是，人口数量的输入框需要字符串类型的值，否则 Mechanize 会抛出 TypeError 异常。

要检查上面的代码是否运行成功，可以使用 Mechanize 返回到国家表单，查询当前的人口数量，如下所示。

```
>>> br.open(COUNTRY_URL)
>>> br.select_form(nr=0)
>>> br['population']
62348449
```

和预期一样，人口数量再次增加，现在是 **62,348,449**。

更多关于 Mechanize 模块的文档和示例可以通过其项目网站获取，网址为 http://wwwsearch.sourceforge.net/mechanize/。

6.4　本章小结

在抓取网页时，和表单进行交互是一个非常重要的技能。本章介绍了两种交互方法：第一种是分析表单，然后手工生成期望的 POST 请求；第二种则是直接使用高级模块 Mechanize。

下一章，我们将会继续扩展表单相关的技能，学习如何提交需要发送验证码的表单。

第 7 章
验证码处理

验证码（CAPTCHA）的全称为**全自动区分计算机和人类的公开图灵测试（Completely Automated Public Turing test to tell Computersand Humans Apart）**。从其全称可以看出，验证码用于测试用户是否为真实人类。一个典型的验证码由扭曲的文本组成，此时计算机程序难以解析，但人类仍然可以（希望如此）阅读。许多网站使用验证码来防御与其网站交互的机器人程序。比如许多银行网站强制每次登录时都需要输入验证码，这就令人十分痛苦。本章将介绍如何自动化处理验证码问题，首先使用**光学字符识别（Optical Character Recognition，OCR）**，然后使用一个验证码处理 API。

7.1 注册账号

在前一章处理表单时，我们使用手工创建的账号登录网站，而忽略了自动化创建账号这一部分，这是因为注册表单需要输入验证码。注册页面为 `http://example.webscraping.com/user/register`，如图 7.1 所示。

请注意，每次加载表单时都会显示不同的验证码图像。为了了解表单需要哪些参数，我们可以复用上一章编写的 `parse_form()` 函数。

```
>>> import cookielib, urllib2, pprint
>>> REGISTER_URL = 'http://example.webscraping.com/user/register'
>>> cj = cookielib.CookieJar()
>>> opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
>>> html = opener.open(REGISTER_URL).read()
>>> form = parse_form(html)
>>> pprint.pprint(form)
{'_formkey': '1ed4e4c4-fbc6-4d82-a0d3-771d289f8661',
 '_formname': 'register',
 '_next': '/',
 'email': '',
 'first_name': '',
 'last_name': '',
 'password': '',
 'password_two': None,
 'recaptcha_response_field': None}
```

前面的代码中，除 recaptcha_response_field 之外的其他域都很容易处理，在本例中这个域要求我们从图像中抽取出 **strange** 字符串。

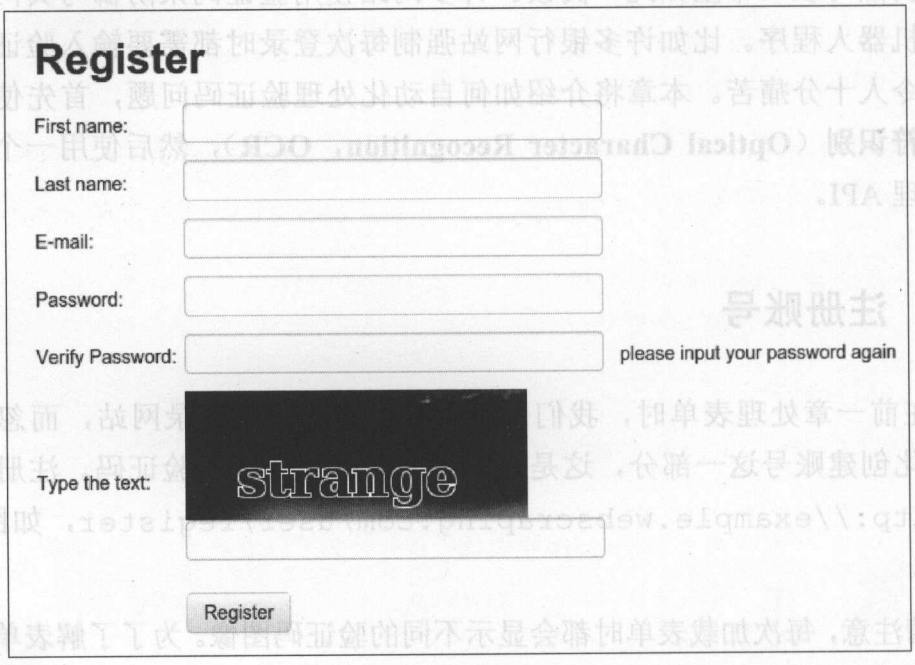


图 7.1

7.1.1 加载验证码图像

在分析验证码图像之前，首先需要从表单中获取该图像。通过 Firebug 可以看到，图像数据是嵌入在网页中的，而不是从其他 URL 加载过来的，如图 7.2 所示。

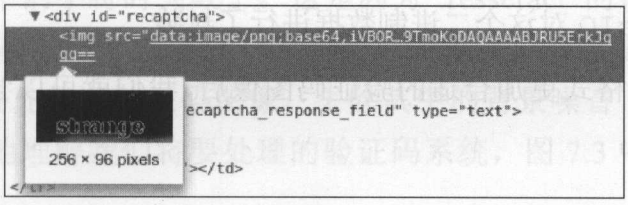


图 7.2

为了在 Python 中处理该图像，我们将会用到 `Pillow` 包，可以使用如下命令通过 `pip` 安装该包。

```
pip install Pillow
```

其他安装 Pillow 的方法可以参考 `http://pillow.readthedocs.org/installation.html`。

`Pillow` 提供了一个便捷的 `Image` 类，其中包含了很多用于处理验证码图像的高级方法。下面的函数使用注册页的 HTML 作为输入参数，返回包含验证码图像的 `Image` 对象。

```
from io import BytesIO
import lxml.html
from PIL import Image

def get_captcha(html):
    tree = lxml.html.fromstring(html)
    img_data = tree.cssselect('div#recaptcha img')[0].get('src')
    img_data = img_data.partition(',')[-1]
    binary_img_data = img_data.decode('base64')
    file_like = BytesIO(binary_img_data)
    img = Image.open(file_like)
    return img
```

开始几行使用 lxml 从表单中获取图像数据。图像数据的前缀定义了数据类型。在本例中，这是一张进行了 Base64 编码的 PNG 图像，这种格式会使用 ASCII 编码表示二进制数据。我们可以通过在第一个逗号处分割的方法移除该前缀。然后，使用 Base64 解码图像数据，回到最初的二进制格式。要想加载图像，PIL 需要一个类似文件的接口，所以在传给 Image 类之前，我们又使用了 BytesIO 对这个二进制数据进行了封装。

在得到这个格式更加合适的验证码图像后，我们就可以尝试从中抽取文本了。

Pillow 与 PIL 的对比

Pillow 是知名的 **Python 图像处理库（Python Image Library，PIL）**的分支版本，不过 PIL 从 2009 年开始就没有再更新过。Pillow 使用了和原始 PIL 包相同的接口，并且拥有完善的文档，其文档地址为 http://pillow.readthedocs.org。本章中的所有示例在 Pillow 和 PIL 中都可以正常运行。

7.2　光学字符识别

光学字符识别（Optical Character Recognition，OCR）用于从图像中抽取文本。本节中，我们将使用开源的 Tesseract OCR 引擎。该引擎最初由惠普公司开发，目前由 Google 主导。Tesseract 的安装说明可以从 https://code.google.com/p/tesseract-ocr/wiki/ReadMe 获取。然后，可以使用 pip 安装其 Python 封装版本 pytesseract。

```
pip install pytesseract
```

如果直接把验证码原始图像传给 pytesseract，解析结果一般都会很糟糕。

```
>>> import pytesseract
>>> img = get_captcha(html)
>>> pytesseract.image_to_string(img)
''
```

上面的代码在执行后，会返回一个空字符串[①]，也就是说 Tesseract 在抽取输入图像中的字符时失败了。这是因为 Tesseract 的设计初衷是抽取更加典型的文本，比如背景统一的书页。如果我们想要更加有效地使用 Tesseract，需要先修改验证码图像，去除其中的背景噪音，只保留文本部分。为了更好地理解我们将要处理的验证码系统，图 7.3 中又给出了几个示例验证码。

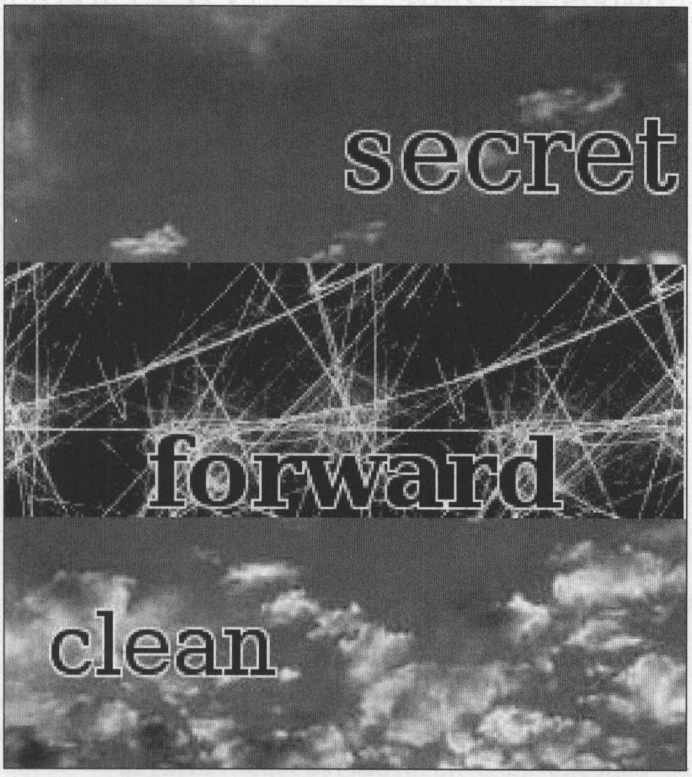


图 7.3

[①] 译者注：返回值也可能是一个错误的解析结果。

从图 7.3 中的例子可以看出，验证码文本一般都是黑色的，背景则会更加明亮，所以我们可以通过检查像素是否为黑色将文本分离出来，该处理过程又被称为**阈值化**。通过 Pillow 可以很容易地实现该处理过程。

```
>>> img.save('captcha_original.png')
>>> gray = img.convert('L')
>>> gray.save('captcha_gray.png')
>>> bw = gray.point(lambda x: 0 if x < 1 else 255, '1')
>>> bw.save('captcha_thresholded.png')
```

此时，只有阈值小于 1 的像素才会保留，也就是说，只有全黑的像素才会保留下来。这段代码片段保存了 3 张图像，分别是原始验证码图像、转换后的灰度图以及阈值化处理后的图像。图 7.4 所示为每个阶段保存的图像。

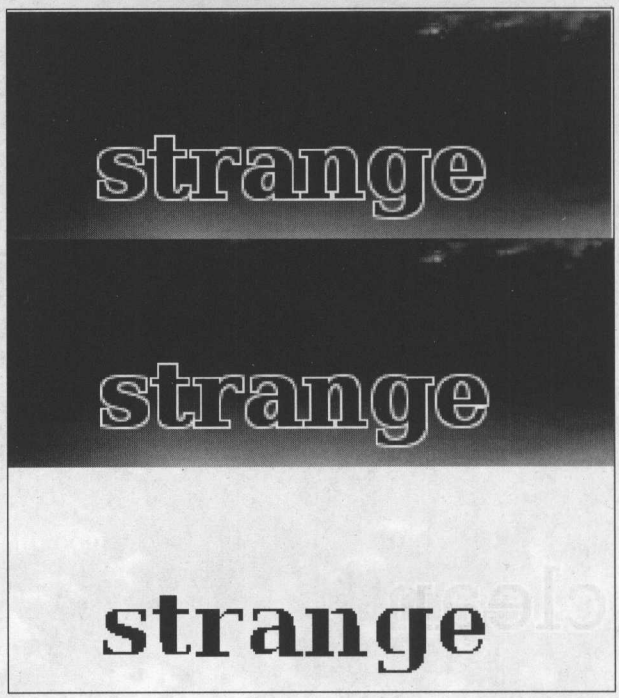


图 7.4

最终，经过阈值化处理的图像中，文本更加清晰，此时我们就可以将其传给 Tesseract 进行处理了。

```
>>> pytesseract.image_to_string(bw)
'strange'
```

成功了！验证码中的文本已经被成功抽取出来了。在我测试的 100 张图片中，该方法正确解析了其中的 84 个验证码图像。由于示例文本总是小写的 ASCII 字符，因此我们可以将结果限定在这些字符中，从而进一步提高性能。

```
>>> import string
>>> word = pytesseract.image_to_string(bw)
>>> ascii_word = ''.join(c for c in word if c in
    string.letters).lower()
```

在对相同的 100 张图片的测试中，其识别率提高到了 88%。下面是目前注册脚本的完整代码。

```
import cookielib
import urllib
import urllib2
import string
import pytesseract
REGISTER_URL = 'http://example.webscraping.com/user/register'

def register(first_name, last_name, email, password):
    cj = cookielib.CookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    html = opener.open(REGISTER_URL).read()
    form = parse_form(html)
    form['first_name'] = first_name
    form['last_name'] = last_name
    form['email'] = email
    form['password'] = form['password_two'] = password
    img = extract_image(html)
    captcha = ocr(img)
    form['recaptcha_response_field'] = captcha
    encoded_data = urllib.urlencode(form)
    request = urllib2.Request(REGISTER_URL, encoded_data)
    response = opener.open(request)
    success = '/user/register' not in response.geturl()
    return success
```

```
def ocr(img):
    gray = img.convert('L')
    bw = gray.point(lambda x: 0 if x < 1 else 255, '1')
    word = pytesseract.image_to_string(bw)
    ascii_word = ''.join(c for c in word if c in
        string.letters).lower()
    return ascii_word
```

register()函数下载注册页面，抓取其中的表单，并在表单中设置新账号的名称、邮箱地址和密码。然后抽取验证码图像，传给 OCR 函数，并将 OCR 函数产生的结果添加到表单中。接下来提交表单数据，检查响应 URL，确认注册是否成功。如果注册失败，响应 URL 仍然会是注册页，这既可能是因为验证码图像解析不正确，也可能是注册账号的邮箱地址已经存在。现在，只需要使用新账号信息调用 register()函数，就可以注册账号了。

```
>>> register(first_name, last_name, email, password)
True
```

7.2.1　进一步改善

要想进一步改善验证码 OCR 的性能，下面还有一些可能会使用到的方法：

- 实验不同阈值；
- 腐蚀阈值文本，突出字符形状；
- 调整图像大小（有时增大尺寸会起到作用）；
- 根据验证码字体训练 OCR 工具；
- 限制结果为字典单词。

如果你对改善性能的实验感兴趣，可以使用该链接中的示例数据：https://bitbucket.org/wswp/code/src/tip/chapter07/samples/。不过，对于我们注册账号这一目的，目前 88%的准确率已经足够了，这是因为即使是真实用户也会在输入验证码文本时出现错误。实际上，即使 1%的准确率也是足够的，因为脚本可以运行多次直至成功，不过这样做对服务器不

够友好，甚至可能会导致 IP 被封禁。

7.3 处理复杂验证码

前面用于测试的验证码系统相对来说比较容易处理，因为文本使用的黑色字体与背景很容易区分，而且文本是水平的，无须旋转就能被 Tesseract 准确解析。一般情况下，网站使用的都是类似这种比较简单的通用验证码系统，此时可以使用 OCR 方法。但是，如果网站使用的是更加复杂的系统，比如 Google 的 reCAPTCHA，OCR 方法则需要花费更多努力，甚至可能无法使用。图 7.5 所示为网络上找到的一些更加复杂的验证码图像。

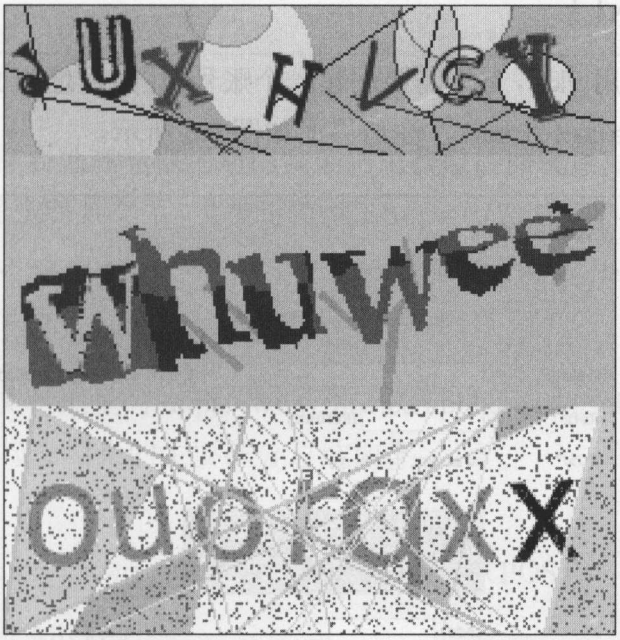


图 7.5

在这些例子中，因为文本被置于不同的角度，并且拥有不同的字体和颜色，所以要使用 OCR 方法的话，需要更多工作来清理这些噪音。即使是真实人类，解析这些图像也可能会存在困难，尤其是对于那些视觉障碍人士而言。

7.3.1　使用验证码处理服务

为了处理这些更加复杂的图像，我们将使用验证码处理服务[①]。验证码处理服务有很多，比如 2captcha.com 和 deathbycaptcha.com，一般其服务价位在 1 美元 1000 个验证码图像左右。当把验证码图像传给它们的 API 时，会有人进行人工查看，并在 HTTP 响应中给出解析后的文本。一般来说该过程在 30 秒以内。在本节的示例中，我们将使用 9kw.eu 的服务。虽然该服务没有提供最便宜的验证码处理价格，也没有最好的 API 设计，但是使用该 API 可能不需要花钱。这是因为 9kw.eu 允许用户人工处理验证码以获取积分，然后花费这些积分处理我们的验证码。

7.3.2　9kw 入门

要想开始使用 9kw，首先需要创建一个账号，注册网址为 https://www.9kw.eu/register.html，注册界面如图 7.6 所示。

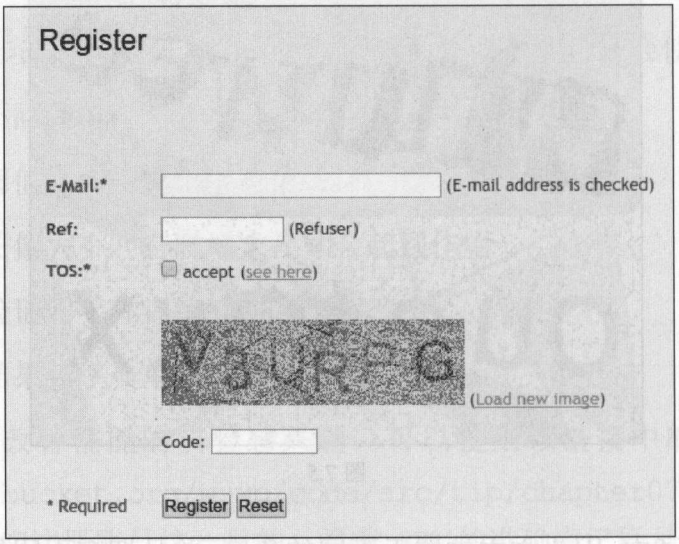


图 7.6

① 译者注：一般也称为打码平台。

然后，按照账号确认说明进行操作。登录后，我们被定位到 https://www.9kw.eu/usercaptcha.html，其页面显示如图 7.7 所示。

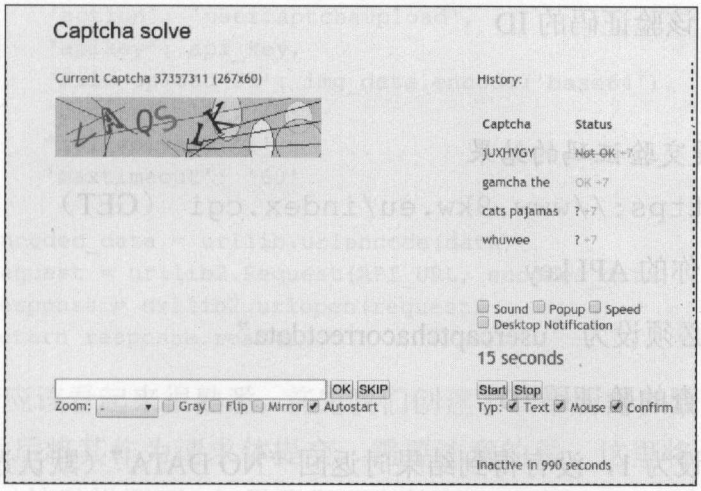


图 7.7

在本页中，需要处理其他用户的验证码来获取后面使用 API 时所需的积分。在处理了几个验证码之后，会被定位到 https://www.9kw.eu/index.cgi?action=userapinew&source=api 来创建 API key。

9kw 验证码 API

9kw 的 API 文档地址为 https://www.9kw.eu/api.html#apisubmit-tab。我们用于提交验证码和检查结果的主要部分总结如下。

提交验证码

URL：https://www.9kw.eu/index.cgi（POST）

apikey：你的 API key

action：必须设为“usercaptchaupload”

file-upload-01：需要处理的图像（文件、url 或字符串）

base64：如果输入是 Base64 编码，则设为“1”

maxtimeout：等待处理的最长时间（必须为 60～3999 秒）

selfsolve：如果自己处理该验证码，则设为“1”

返回值：该验证码的 ID

请求已提交验证码的结果

URL：`https://www.9kw.eu/index.cgi` （GET）

apikey：你的 API key

action：必须设为“usercaptchacorrectdata”

id：要检查的验证码 ID

info：若设为 1，没有得到结果时返回“NO DATA”（默认返回空）

返回值：要处理的验证码文本或错误码

错误码

0001 API key 不存在

0002 没有找到 API key

0003 没有找到激活的 API key

……

0031 账号被系统禁用 24 小时

0032 账号没有足够的权限

0033 需要升级插件

下面是发送验证码图像到该 API 的初始实现代码。

```
import urllib
import urllib2
```

```
API_URL = 'https://www.9kw.eu/index.cgi'

def send_captcha(api_key, img_data):
    data = {
        'action': 'usercaptchaupload',
        'apikey': api_key,
        'file-upload-01': img_data.encode('base64'),
        'base64': '1',
        'selfsolve': '1',
        'maxtimeout': '60'
    }
    encoded_data = urllib.urlencode(data)
    request = urllib2.Request(API_URL, encoded_data)
    response = urllib2.urlopen(request)
    return response.read()
```

这个结构应该看起来很熟悉，首先我们创建了一个所需参数的字典，对其进行编码，然后将其作为请求体提交。需要注意的是，这里将 selfsolve 选项设为'1'，这种设置下，如果我们正在使用 9kw 的 Web 界面处理验证码，那么验证码图像就会传给我们自己处理，从而可以节约我们的积分。如果此时我们没有处于登录状态，验证码则会像平时一样传给其他用户。

下面是获取验证码图像处理结果的代码。

```
def get_captcha(api_key, captcha_id):
    data = {
        'action': 'usercaptchacorrectdata',
        'id': captcha_id,
        'apikey': api_key
    }
    encoded_data = urllib.urlencode(data)
    # note this is a GET request
    # so the data is appended to the URL
    response = urllib2.urlopen(API_URL + '?' + encoded_data)
    return response.read()
```

9kw 的 API 有一个缺点是其响应是普通字符串，而不是类似 JSON 的结构化格式，这样就会使错误信息的区分更加复杂。例如，此时没有用户处理验证码图像，则会返回 ERROR NO USER 字符串。不过幸好我们提交的验证码图像永远不会包含这类文本。

另一个困难是，只有在其他用户有时间人工处理验证码图像时，get_captcha()函数才能返回错误信息，正如之前提到的，通常要在 30 秒之后。为了使实现更加友好，我们将会增加一个封装函数，用于提交验证码图像以及等待结果返回。下面的扩展版本代码把这些功能封装到一个可复用类当中，另外还增加了检查错误信息的功能。

```
import time
import urllib
import urllib2
import re
from io import BytesIO

class CaptchaAPI:
    def __init__(self, api_key, timeout=60):
        self.api_key = api_key
        self.timeout = timeout
        self.url = 'https://www.9kw.eu/index.cgi'

    def solve(self, img):
        """Submit CAPTCHA and return result when ready
        """
        img_buffer = BytesIO()
        img.save(img_buffer, format="PNG")
        img_data = img_buffer.getvalue()
        captcha_id = self.send(img_data)
        start_time = time.time()
        while time.time() < start_time + self.timeout:
            try:
                text = self.get(captcha_id)
            except CaptchaError:
                pass # CAPTCHA still not ready
            else:
                if text != 'NO DATA':
                    if text == 'ERROR NO USER':
                        raise CaptchaError('Error: no user
                            available to solve CAPTCHA')
                    else:
                        print 'CAPTCHA solved!'
```

```
                return text
            print 'Waiting for CAPTCHA ...'

        raise CaptchaError('Error: API timeout')

    def send(self, img_data):
        """Send CAPTCHA for solving
        """
        print'Submitting CAPTCHA'
        data = {
            'action': 'usercaptchaupload',
            'apikey': self.api_key,
            'file-upload-01': img_data.encode('base64'),
            'base64': '1',
            'selfsolve': '1',
            'maxtimeout': str(self.timeout)
        }
        encoded_data = urllib.urlencode(data)
        request = urllib2.Request(self.url, encoded_data)
        response = urllib2.urlopen(request)
        result = response.read()
        self.check(result)
        return result

    def get(self, captcha_id):
        """Get result of solved CAPTCHA
        """
        data = {
            'action': 'usercaptchacorrectdata',
            'id': captcha_id,
            'apikey': self.api_key,
            'info': '1'
        }
        encoded_data = urllib.urlencode(data)
        response = urllib2.urlopen(self.url + '?' + encoded_data)
        result = response.read()
        self.check(result)
        return result

    def check(self, result):
```

```
        """Check result of API and raise error if error code
        """
        if re.match('00\d\d \w+', result):
            raise CaptchaError('API error: ' + result)

class CaptchaError(Exception):
    pass
```

CaptchaAPI 类的源码可以从 https://bitbucket.org/wswp/code/src/tip/chapter07/api.py 获取，该链接中的代码会在 9kw.eu 修改其 API 时保持更新。这个类使用你的 API key 以及超时时间进行实例化，其中超时时间默认为 60 秒。然后，solve()方法把验证码图像提交给 API，并持续请求，直到验证码图像处理完成或者到达超时时间。目前，检查 API 响应中的错误信息时，check()方法只检查了初始字符，确认其是否遵循错误信息前包含 4 位数字错误码的格式。要想该 API 在使用时更加健壮，可以对该方法进行扩展，使其包含全部 34 种错误类型。

下面是使用 CaptchaAPI 类处理验证码图像时的执行过程示例。

```
>>> API_KEY = ...
>>> captcha = CaptchaAPI(API_KEY)
>>> img = Image.open('captcha.png')
>>> text = captcha.solve(img)
Submitting CAPTCHA
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
CAPTCHA solved!
>>> text
juxhvgy
```

这是本章前面给出的第一个复杂验证码图像的正确识别结果。如果再次提交相同的验证码图像，则会立即返回缓存结果，并且不会再次消耗积分。

```
>>> text = captcha.solve(img_data)
Submitting CAPTCHA
>>> text
juxhvgy
```

7.3.3 与注册功能集成

目前我们已经拥有了一个可以运行的验证码 API 解决方案，下面我们可以将其与前面的表单进行集成。下面的代码对 register 函数进行了修改，使其将处理验证码图像的函数作为参数传递进来，这样我们就可以选择使用 OCR 方法还是 API 方法了。

```
def register(first_name, last_name, email, password, captcha_fn):
    cj = cookielib.CookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    html = opener.open(REGISTER_URL).read()
    form = parse_form(html)
    form['first_name'] = first_name
    form['last_name'] = last_name
    form['email'] = email
    form['password'] = form['password_two'] = password
    img = extract_image(html)
    form['recaptcha_response_field'] = captcha_fn(img)
    encoded_data = urllib.urlencode(form)
    request = urllib2.Request(REGISTER_URL, encoded_data)
    response = opener.open(request)
    success = '/user/register' not in response.geturl()
    return success
```

下面是使用该函数的例子。

```
>>> captcha = CaptchaAPI(API_KEY)
>>> register(first_name, last_name, email, password, captcha.solve)
Submitting CAPTCHA
Waiting for CAPTCHA ...
```

```
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
True
```

运行成功了！我们从表单中成功获取到验证码图像，提交给 9kw 的 API，之后其他用户人工处理了该验证码，程序将返回结果提交到 Web 服务器端，注册了一个新账号。

7.4　本章小结

本章给出了处理验证码的方法：首先是使用 OCR，然后是使用外部 API。对于简单的验证码，或者需要处理大量验证码时，在 OCR 方法上花费时间是很值得的。否则，使用验证码处理 API 会更加经济有效。

下一章中，我们将介绍 Scrapy，这是一个流行的高级框架，可以用于创建抓取应用。

第8章 Scrapy

Scrapy 是一个流行的网络爬虫框架，它拥有很多简化网站抓取的高级函数。本章中，我们将学习使用 Scrapy 抓取示例网站，目标任务与第 2 章相同。然后，我们还会介绍 **Portia**，这是一个基于 Scrapy 的应用，允许用户通过点击界面抓取网站。

8.1 安装

我们可以使用 `pip` 命令安装 Scrapy，如下所示。

```
pip install Scrapy
```

由于 Scrapy 依赖一些外部库，因此如果在安装过程中遇到困难的话，可以从其官方网站上获取到更多信息，网址为 `http://doc.scrapy.org/en/latest/intro/install.html`。

目前，Scrapy 仅支持 Python 2.7 版本[①]，比本书中介绍的其他包条件更加苛刻。如果想支持更低的 Python 2.6 版本，需要降级到 Scrapy 0.20 版本。而由于依赖的 Twisted 的原因，目前还无法支持 Python 3 版本，不过 Scrapy 团

[①] 译者注：Scrapy 从 1.1.0 版本开始加入对 Python 3 的支持，但目前仍处于试验阶段，部分功能还未得到有效支持。Scrapy 1.1.0 版本需要 Python 3.3 及以上版本、Twisted 15.5 及以上版本。详细信息可以参考 Scrapy 的官方文档：`http://doc.scrapy.org/en/latest/news.html#beta-python-3-support`。

队向我确认他们正在解决这一问题。

如果 Scrapy 安装成功，那么就可以在终端里执行 scrapy 命令了。

```
$ scrapy -h
Scrapy 0.24.4 - no active project

Usage:
    scrapy <command> [options] [args]

Available commands:
    bench         Run quick benchmark test
    check         Check spider contracts
    crawl         Run a spider
...
```

本章中我们将会使用如下几个命令。

- startproject：创建一个新项目；
- genspider：根据模板生成一个新爬虫；
- crawl：执行爬虫；
- shell：启动交互式抓取控制台。

要了解上述命令或其他命令的详细信息，可以参考 http://doc.scrapy.org/en/latest/topics/commands.html。

8.2　启动项目

安装好 Scrapy 以后，我们可以运行 startproject 命令生成该项目的默认结构。具体步骤为：打开终端进入想要存储 Scrapy 项目的目录，然后运行 scrapy startproject <project name>。这里我们使用 example 作为项目名。

```
$ scrapy startproject example
$ cd example
```

下面是 scrapy 命令生成的文件结构。

```
scrapy.cfg
example/
    __init__.py
    items.py
    pipelines.py
    settings.py
    spiders/
        __init__.py
```

其中，在本章比较重要的几个文件如下所示。

- items.py：该文件定义了待抓取域的模型。
- settings.py：该文件定义了一些设置，如用户代理、爬取延时等。
- spiders/：该目录存储实际的爬虫代码。

另外，Scrapy 使用 scrapy.cfg 设置项目配置，使用 pipelines.py 处理要抓取的域，不过在本例中无须修改这两个文件。

8.2.1 定义模型

默认情况下，example/items.py 文件包含如下代码。

```
import scrapy

class ExampleItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    pass
```

ExampleItem 类是一个模板，需要将其中的内容替换为爬虫运行时想要存储的待抓取国家信息。为了更好地聚焦 Scrapy 的执行过程，接下来我们只会抓取国家名称和人口数量，而不是抓取国家的所有信息。下面是修改后支

持该功能的模型代码。

```
class ExampleItem(scrapy.Item):
    name = scrapy.Field()
    population = scrapy.Field()
```

定义模型的详细文档可以参考 http://doc.scrapy.org/en/latest/topics/ items.html。

8.2.2　创建爬虫

现在，我们要开始编写真正的爬虫代码了，在 Scrapy 里又被称为 **spider**。通过 genspider 命令，传入爬虫名、域名以及可选的模板参数，就可以生成初始模板。

```
$ scrapy genspider country example.webscraping.com --template=crawl
```

这里使用内置的 crawl 模板，可以生成更接近我们想要的国家爬虫的初始版本。运行 genspider 命令之后，下面的代码将会在 example/spiders/country.py 中自动生成。

```
import scrapy
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

from example.items import ExampleItem

class CountrySpider(CrawlSpider):
    name = 'country'
    start_urls = ['http://www.example.webscraping.com/']
    allowed_domains = ['example.webscraping.com']

    rules = (
        Rule(LinkExtractor(allow=r'Items/'),
             callback='parse_item', follow=True),
```

```
    )

    def parse_item(self, response):
        i = ExampleItem()
        #i['domain_id'] =
            response.xpath('//input[@id="sid"]/@value').extract()
        #i['name'] = response.xpath('//div[@id="name"]').extract()
        #i['description'] =
            response.xpath('//div[@id="description"]').extract()
        return i
```

最开始几行导入了后面会用到的 Scrapy 库，包括 8.2.1 节中定义的 ExampleItem 模型。然后创建了一个爬虫类，该类包括如下类属性。

- name：该属性为定义爬虫名称的字符串。
- start_urls：该属性定义了爬虫起始URL列表。不过，start_urls 的默认值与我们想要的不一样，在 example.webscraping.com 域名之前多了 www 前缀。
- allowed_domains：该属性定义了可以爬取的域名列表。如果没有定义该属性，则表示可以爬取任何域名。
- rules：该属性为一个正则表达式集合，用于告知爬虫需要跟踪哪些链接。

rules 属性还有一个 callback 函数，用于解析下载得到的响应，而 parse_item()示例方法给我们提供了一个从响应中获取数据的例子。

Scrapy 是一个高级框架，因此即使只有这几行代码，也还有很多需要了解的知识。官方文档中包含了创建爬虫相关的更多细节，其网址为 http://doc.scrapy.org/en/latest/topics/spiders.html。

1. 优化设置

在运行前面生成的爬虫之前，需要更新 Scrapy 的设置，避免爬虫被封禁。默认情况下，Scrapy 对同一域名允许最多 8 个并发下载，并且两次下载之间

没有延时，这样就会比真实用户浏览时的速度快很多，所以很容易被服务器检测到。在前言中我们提到，当下载速度持续高于每秒一个请求时，我们抓取的示例网站会暂时封禁爬虫，也就是说使用默认配置会造成我们的爬虫被封禁。除非你在本地运行示例网站，否则我建议在 example/settings.py 文件中添加如下几行，使爬虫同时只能对每个域名发起一个请求，并且每两次请求之间存在延时：

```
CONCURRENT_REQUESTS_PER_DOMAIN = 1
DOWNLOAD_DELAY = 5
```

请注意，Scrapy 在两次请求之间的延时并不是精确的，这是因为精确的延时同样会造成爬虫容易被检测到，然后被封禁。而 Scrapy 实际使用的方法是在两次请求之间的延时上添加随机的偏移量。要想了解更多关于上述设置和其他设置的细节，可以参考 http://doc.scrapy.org/en/latest/topics/ settings.html。

2. 测试爬虫

想要从命令行运行爬虫，需要使用 crawl 命令，并且带上爬虫的名称。

```
$ scrapy crawl country -s LOG_LEVEL=ERROR
[country] ERROR: Error downloading <GET http://www.example.webscraping.
com/>: DNS lookup failed: address 'www.example.webscraping.com' not
found: [Errno -5] No address associated with hostname.
```

和预期一样，默认的爬虫代码运行失败了，这是因为 http://www.example.webscraping.com 并不存在[①]。此外，你还会注意到命令中有一个 -s LOG_LEVEL=ERROR 标记，这是一个 Scrapy 设置，等同于在 settings.py 文件中定义 LOG_LEVEL = 'ERROR'。默认情况下，Scrapy 会在终端上输出所有日志信息，而这里是将日志级别提升至只显示错误信息。

[①] 译者注：在本书翻译时，该域名已经配置了 DNS 解析，此时返回结果应该为空，而不是错误信息。

下面的代码更正了爬虫的起始 URL，并且设定了要爬取的网页。

```
start_urls = ['http://example.webscraping.com/']

rules = (
    Rule(LinkExtractor(allow='/index/'), follow=True),
    Rule(LinkExtractor(allow='/view/'), callback='parse_item')
)
```

第一条规则爬取索引页并跟踪其中的链接，而第二条规则爬取国家页面并将下载响应传给 callback 函数用于抓取。下面让我们把日志级别设为 DEBUG 以显示所有信息，来看下爬虫是如何运行的。

```
$ scrapy crawl country -s LOG_LEVEL=DEBUG
...
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/>
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/index/1>
[country] DEBUG: Filtered duplicate request: <GET http://example.webscraping.
com/ index/1> - no more duplicates will be shown (see DUPEFILTER_DEBUG
to show all duplicates)
[country] DEBUG: Crawled (200) <GET
http://example.webscraping.com/view/Antigua-and-Barbuda-10>
[country] DEBUG: Crawled (200) <GET
http://example.webscraping.com/user/login?_next=%2Findex%2F1>
[country] DEBUG: Crawled (200) <GET
http://example.webscraping.com/user/register?_next=%2Findex%2F1>
...
```

输出的日志信息显示，索引页和国家页都可以正确爬取，并且已经过滤了重复链接。但是，我们还会发现爬虫浪费了很多资源来爬取每个网页上的登录和注册表单链接，因为它们也匹配 rules 里的正则表达式。前面命令中的登录 URL 以 _next=%2Findex%2F1 结尾，也就是 _next=/index/1 经过 URL 编码后的结果，其目的是让服务器端获取用户登录后的跳转地址。要想避免爬取这些 URL，我们可以使用规则的 deny 参数，该参数同样需要一个正则表达式，用于匹配所有不想爬取的 URL。下面对之前的代码进行了修改，通过避免 URL 包含/user/来防止爬取用户登录和注册表单。

```
    rules = (
        Rule(LinkExtractor(allow='/index/', deny='/user/'),
             follow=True),
        Rule(LinkExtractor(allow='/view/', deny='/user/'),
             callback='parse_item')
    )
```

想要进一步了解如何使用该类，可以参考其文档，网址为 http://doc.scrapy.org/en/latest/topics/linkextractors.html。

8.2.3　使用 shell 命令抓取

现在 Scrapy 已经可以爬取国家页面了，下面还需要定义要抓取哪些数据。为了帮助测试如何从网页中抽取数据，Scrapy 提供了一个很方便的命令——shell，可以下载 URL 并在 Python 解释器中给出结果状态。下面是爬取某个示例国家时的结果。

```
$ scrapy shell http://example.webscraping.com/view/United-Kingdom-239
[s] Available Scrapy objects:
[s] crawler <scrapy.crawler.Crawler object at 0x7f1475da5390>
[s] item {}
[s] request <GET http://example.webscraping.com/view/United-Kingdom-239>
[s] response <200 http://example.webscraping.com/view/United-Kingdom-239>
[s] settings <scrapy.settings.Settings object at 0x7f147c1fb490>
[s] spider <Spider 'default' at 0x7f147552eb90>
[s] Useful shortcuts:
[s] shelp() Shell help (print this help)
[s] fetch(req_or_url) Fetch request (or URL) and update local objects
[s] view(response) View response in a browser
```

现在我们可以查询这些对象，检查哪些数据可以使用。

```
In [1]: response.url
'http://example.webscraping.com/view/United-Kingdom-239'
In [2]: response.status
200
```

Scrapy 使用 lxml 抓取数据，所以我们仍然可以使用第 2 章中用过的 CSS 选择器。

```
In [3]: response.css('tr#places_country__row td.w2p_fw::text')
 [<Selector xpath=u"descendant-or-self::
    tr[@id = 'places_country__row']/descendant-or-self::
    */td[@class and contains(
    concat(' ', normalize-space(@class), ' '),
    ' w2p_fw ')]/text()" data=u'United Kingdom'>]
```

该方法返回一个 lxml 选择器，要想使用它，还需要调用 extract() 方法。

```
In [4]: name_css = 'tr#places_country__row td.w2p_fw::text'
In [5]: response.css(name_css).extract()
[u'United Kingdom']
In [6]: pop_css = 'tr#places_population__row td.w2p_fw::text'
In [7]: response.css(pop_css).extract()
 [u'62,348,447']
```

然后，可以在先前生成的 example/spiders/country.py 文件的 parse_item() 方法中使用这些 CSS 选择器。

```
def parse_item(self, response):
    item = ExampleItem()
    name_css = 'tr#places_country__row td.w2p_fw::text'
    item['name'] = response.css(name_css).extract()
    pop_css = 'tr#places_population__row td.w2p_fw::text'
    item['population'] = response.css(pop_css).extract()
    return item
```

8.2.4 检查结果

下面是该爬虫的完整代码。

```
class CountrySpider(CrawlSpider):
    name = 'country'
    start_urls = ['http://example.webscraping.com/']
    allowed_domains = ['example.webscraping.com']
    rules = (
```

```
            Rule(LinkExtractor(allow='/index/', deny='/user/'),
                 follow=True),
            Rule(LinkExtractor(allow='/view/', deny='/user/'),
                 callback='parse_item')
        )

    def parse_item(self, response):
        item = ExampleItem()
        name_css = 'tr#places_country__row td.w2p_fw::text'
        item['name'] = response.css(name_css).extract()
        pop_css = 'tr#places_population__row td.w2p_fw::text'
        item['population'] = response.css(pop_css).extract()
        return item
```

要想保存结果，我们可以在 parse_item() 方法中添加额外的代码，用于写入已抓取的国家数据，或是定义管道。不过，这一操作并不是必需的，因为 Scrapy 还提供了一个更方便的--output 选项，用于自动保存已抓取的条目，可选格式包括 CSV、JSON 和 XML。下面是该爬虫的最终版运行时的结果，该结果将会输出到一个 CSV 文件中，此外该爬虫的日志级别被设定为 INFO 以过滤不重要的信息。

```
$ scrapy crawl country --output=countries.csv -s LOG_LEVEL=INFO
[scrapy] INFO: Scrapy 0.24.4 started (bot: example)
[country] INFO: Spider opened
[country] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)
[country] INFO: Crawled 10 pages (at 10 pages/min), scraped 9 items (at 9 items/min)
...
[country] INFO: Crawled 264 pages (at 10 pages/min), scraped 238 items
(at 9 items/min)
[country] INFO: Crawled 274 pages (at 10 pages/min), scraped 248 items
(at 10 items/min)
[country] INFO: Closing spider (finished)
[country] INFO: Stored csv feed (252 items) in: countries.csv
[country] INFO: Dumping Scrapy stats:
  {'downloader/request_bytes': 155001,
   'downloader/request_count': 279,
   'downloader/request_method_count/GET': 279,
   'downloader/response_bytes': 943190,
   'downloader/response_count': 279,
```

```
    'downloader/response_status_count/200': 279,
    'dupefilter/filtered': 61,
    'finish_reason': 'finished',
    'item_scraped_count': 252,
    'log_count/INFO': 36,
    'request_depth_max': 26,
    'response_received_count': 279,
    'scheduler/dequeued': 279,
    'scheduler/dequeued/memory': 279,
    'scheduler/enqueued': 279,
    'scheduler/enqueued/memory': 279}
[country] INFO: Spider closed (finished)
```

在爬取过程的最后阶段，Scrapy 会输出一些统计信息，给出爬虫运行的一些指标。从统计结果中，我们可以了解到爬虫总共爬取了 279 个网页，并抓取到其中的 252 个条目，这与数据库中的国家数量一致，因此我们知道爬虫已经找到了所有国家数据。

要想验证抓取的这些国家信息正确与否，我们可以检查 `countries.csv` 文件中的内容。

```
name,population
Afghanistan,"29,121,286"
Antigua and Barbuda,"86,754"
Antarctica,0
Anguilla,"13,254"
Angola,"13,068,161"
Andorra,"84,000"
American Samoa,"57,881"
Algeria,"34,586,184"
Albania,"2,986,952"
Aland Islands,"26,711"
...
```

和预期一样，表格中包含了每个国家的名称和人口数量。抓取这些数据所要编写的代码比第 2 章中的原始爬虫要少很多，这是因为 Scrapy 提供了很多高级功能。在下一节中，我们将使用 Portia 重新实现该爬虫，而且要编写的代码更少。

8.2.5 中断与恢复爬虫

在抓取网站时，暂停爬虫并于稍后恢复而不是重新开始，有时会很有用。比如，软件更新后重启计算机，或是要爬取的网站出现错误需要稍后继续爬取时，都可能会中断爬虫。非常方便的是，Scrapy 内置了对暂停与恢复爬取的支持，这样我们就不需要再修改示例爬虫了。要开启该功能，我们只需要定义用于保存爬虫当前状态目录的 JOBDIR 设置即可。需要注意的是，多个爬虫的状态需要保存在不同的目录当中。下面是在我们的爬虫中使用该功能的示例。

```
$ scrapy crawl country -s LOG_LEVEL=DEBUG -s JOBDIR=crawls/country
...
[country] DEBUG: Scraped from <200
http://example.webscraping.com/view/Afghanistan-1>
    {'name': [u'Afghanistan'], 'population': [u'29,121,286']}
^C [scrapy] INFO: Received SIGINT, shutting down gracefully. Send again to force
[country] INFO: Closing spider (shutdown)
[country] DEBUG: Crawled (200) <GET
http://example.webscraping.com/view/Antigua-and-Barbuda-10> (referer:
http://example.webscraping.com/)
[country] DEBUG: Scraped from <200
http://example.webscraping.com/view/Antigua-and-Barbuda-10>
    {'name': [u'Antigua and Barbuda'], 'population': [u'86,754']}
[country] DEBUG: Crawled (200) <GET
http://example.webscraping.com/view/Antarctica-9> (referer:
http://example.webscraping.com/)
[country] DEBUG: Scraped from <200
 http://example.webscraping.com/view/Antarctica-9>
    {'name': [u'Antarctica'], 'population': [u'0']}
...
[country] INFO: Spider closed (shutdown)
```

从上述执行过程可以看出，我们使用^C（Ctrl+C）发送终止信号，然后爬虫又完成了几个条目的处理之后才终止。想要 Scrapy 保存爬虫状态，就必须等待它正常结束，而不能经受不住诱惑再次按下 Ctrl+C 强行立即终止！现在，爬虫状态保存在 crawls/country 目录中，之后可以运行同样的命令恢复爬虫运行。

```
$ scrapy crawl country -s LOG_LEVEL=DEBUG -s JOBDIR=crawls/country
...
[country] INFO: Resuming crawl (12 requests scheduled)
[country] DEBUG: Crawled (200) <GET
http://example.webscraping.com/view/Anguilla-8> (referer:
http://example.webscraping.com/)
[country] DEBUG: Scraped from <200
http://example.webscraping.com/view/Anguilla-8>
    {'name': [u'Anguilla'], 'population': [u'13,254']}
[country] DEBUG: Crawled (200) <GET
http://example.webscraping.com/view/Angola-7> (referer:
http://example.webscraping.com/)
[country] DEBUG: Scraped from <200
http://example.webscraping.com/view/Angola-7>
    {'name': [u'Angola'], 'population': [u'13,068,161']}
...
```

此时，爬虫从刚才暂停的地方恢复运行，和正常启动一样继续进行爬取。该功能对于我们的示例网站而言用处不大，因为要下载的页面数量非常小。不过，对于那些需要爬取几个月的大型网站而言，能够暂停和恢复爬虫就非常方便了。

需要注意的是，有一些边界情况在这里没有覆盖，可能会在恢复爬取时产生问题，比如 cookie 过期等，此类问题可以从 Scrapy 的官方文档中进行详细了解，其网址为 http://doc.scrapy.org/en/latest/topics/jobs.html。

8.3 使用 Portia 编写可视化爬虫

Portia 是一款基于 Scrapy 开发的开源工具，该工具可以通过点击要抓取的网页部分来创建爬虫，这样就比手工创建 CSS 选择器的方式更加方便。

8.3.1 安装

Portia 是一款非常强大的工具，为了实现其功能需要依赖很多外部库。由于该工具相对较新，因此下面会稍微介绍一下它的安装步骤。如果未来该工

具的安装步骤有所简化，可以从其最新文档中获取安装方法，网址为 https://github.com/scrapinghub/portia#running-portia。

推荐安装方式的第一步是使用 virtualenv 创建一个虚拟 Python 环境。这里我们将该环境命名为 portia_example，当然你也可以将其替换成其他任何名称。

```
$ pip install virtualenv
$ virtualenv portia_example --no-site-packages
$ source portia_example/bin/activate
(portia_example)$ cd portia_example
```

为什么使用 virtualenv?

假如你的项目使用早期版本的 lxml 进行开发，而新版本的 lxml 引入了一些向后不兼容的变更，会影响你项目的正常运行。但是，其他一些项目准备依赖新版本的 lxml。如果你的项目使用系统中安装的 lxml，那么当 lxml 更新为支持其他项目的新版本时，你的项目将会运行失败。

Ian Bicking 的 virtualenv 提供了一个聪明的办法来解决这一问题，那就是复制系统 Python 可执行文件及其依赖到一个本地目录，创建隔离的 Python 环境。这样就允许项目在本地安装特定的 Python 库版本，独立于外部系统。详细信息可以查看其文档，网址为 https://virtualenv.pypa.io。

然后，在 virtualenv 中安装 Portia 及其依赖。

```
(portia_example)$ git clone https://github.com/scrapinghub/portia
(portia_example)$ cd portia
(portia_example)$ pip install -r requirements.txt
(portia_example)$ pip install -e ./slybot
```

Portia 目前处于活跃开发期，因此在你阅读本书时其接口可能已经发生变化。如果你想使用和本书相同的版本进行开发，可以运行如下 git 命令。

```
(portia_example)$ git checkout 8210941
```

如果你还没有安装 git，可以直接下载 Portia 的最新版，其网址为 https://github.com/scrapinghub/portia/archive/master.zip。

安装完成后，可以进入 slyd 目录运行服务器端来启动 Portia。

```
(portia_example)$ cd slyd
(portia_example)$ twistd -n slyd
```

如果安装成功，就可以在浏览器中访问到 Portia 工具，网址为 http://localhost:9001/static/main.html。

图 8.1 所示为其初始屏幕。

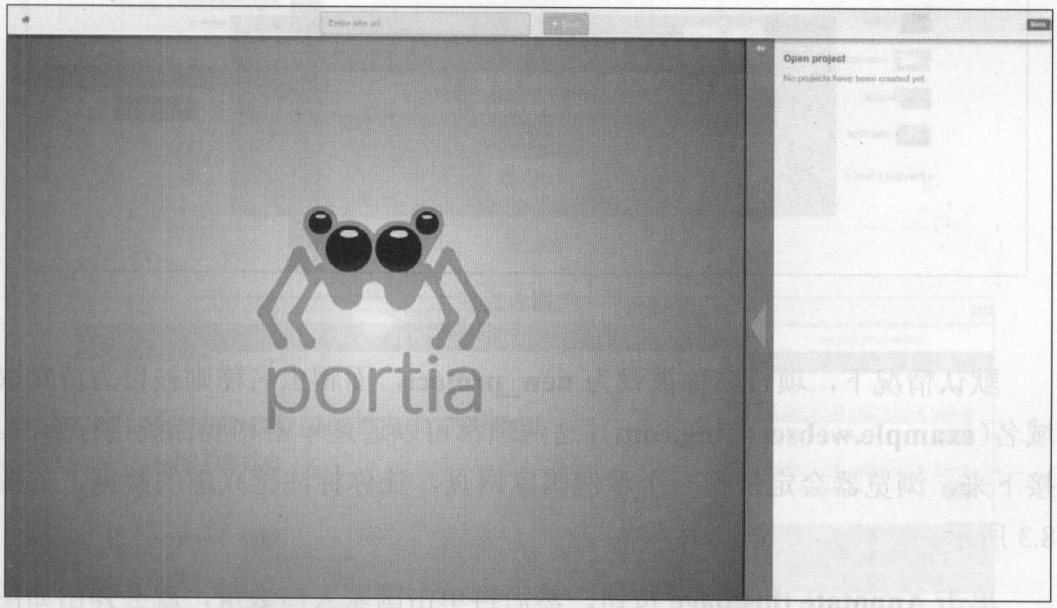


图 8.1

如果你在安装过程中遇到了问题，可以查看 Portia 的问题页，网址为 https://github.com/scrapinghub/portia/issues，也许其他人已经经历过相同的问题并且找到了解决方案。

8.3.2 标注

在 Portia 的启动页，有一个用于输入待抓取网站 URL 的文本框，比如 `http://example.webscraping.com`。输入后，Portia 会在其主面板加载该网页，如图 8.2 所示。

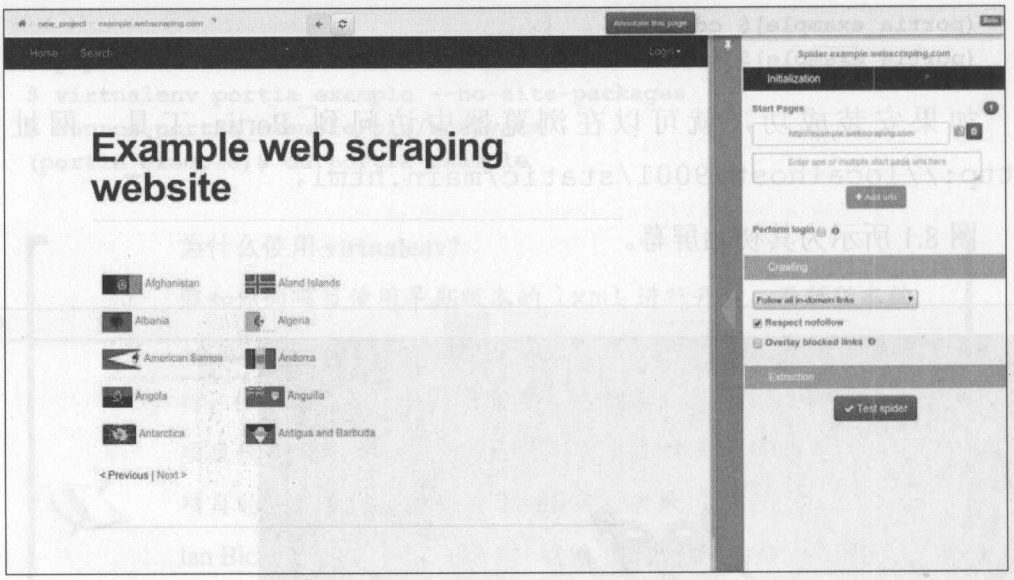


图 8.2

默认情况下，项目名称被设为 **new_project**，而爬虫名称则被设为待爬取域名（**example.webscraping.com**），这两项都可以通过单击相应标签进行修改。接下来，浏览器会定位到一个示例国家网页，让你标注感兴趣的数据，如图 8.3 所示。

单击 **Annotate this page** 按钮，然后再单击国家人口数量，就会弹出如图 8.4 所示的对话框。

单击**+field** 按钮创建一个名为 `population` 的新域，然后单击 **Done** 保存。接下来，对国家名称以及你感兴趣的其他域进行相同操作。被标注的域会在网页中高亮显示，并且可以在右边栏的面板中进行编辑，如图 8.5 所示。

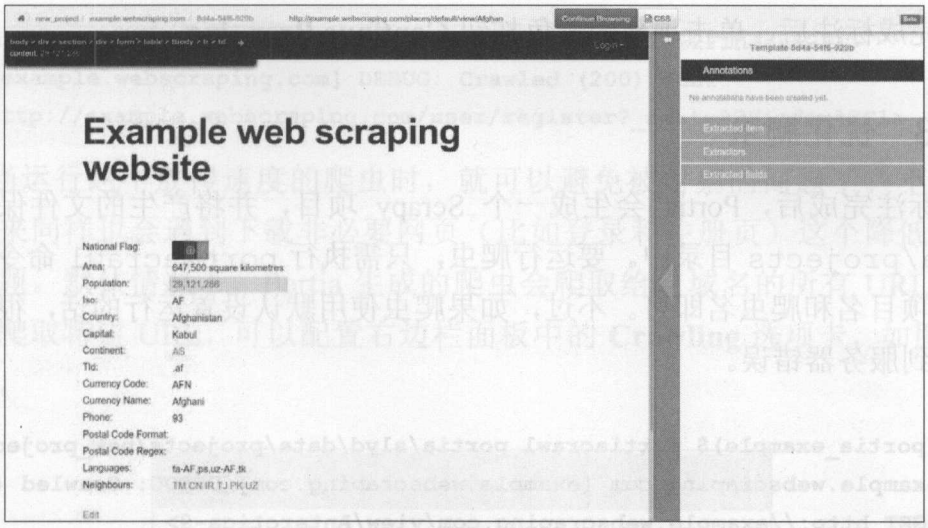

Example web scraping website
National Flag:
Area: 647,500 square kilometres
Population: 29,121,286
Iso: AF
Country: Afghanistan
Capital: Kabul
Continent: AS
Tld: .af
Currency Code: AFN
Currency Name: Afghani
Phone: 93
Postal Code Format:
Postal Code Regex:
Languages: fa-AF,ps,uz-AF,tk
Neighbours: TM CN IR TJ PK UZ
Edit
Annotations
Extracted item
Extractors
Extracted fields

图 8.3


Map attribute
To field
content
-select field-
+ field
Attribute value: 29,121,286
Done

图 8.4

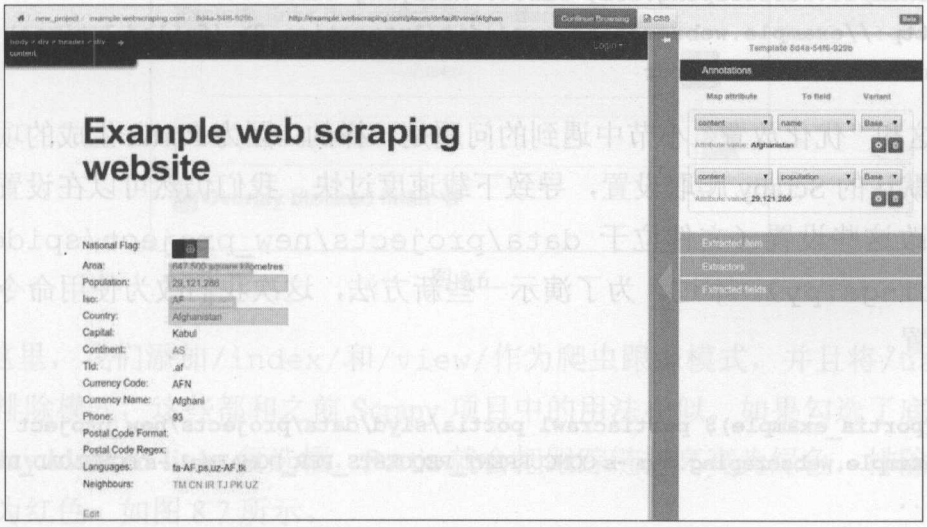

Example web scraping website
National Flag:
Area: 647,500 square kilometres
Population: 29,121,286
Iso: AF
Country: Afghanistan
Capital: Kabul
Continent: AS
Tld: .af
Currency Code: AFN
Currency Name: Afghani
Phone: 93
Postal Code Format:
Postal Code Regex:
Languages: fa-AF,ps,uz-AF,tk
Neighbours: TM CN IR TJ PK UZ
Edit
Annotations
Map attribute
To field
Variant
Extracted item
Extractors
Extracted fields

图 8.5

完成标注后，单击顶部的蓝色按钮 **Continue Browsing**。

8.3.3 优化爬虫

标注完成后，Portia 会生成一个 Scrapy 项目，并将产生的文件保存到 data/projects 目录中。要运行爬虫，只需执行 portiacrawl 命令，并带上项目名和爬虫名即可。不过，如果爬虫使用默认设置运行的话，很快就会遇到服务器错误。

```
(portia_example)$ portiacrawl portia/slyd/data/projects/new_project
example.webscraping.com [example.webscraping.com] DEBUG: Crawled (200)
<GET http://example.webscraping.com/view/Antarctica-9>
[example.webscraping.com] DEBUG: Scraped from <200
http://example.webscraping.com/view/Antarctica-9>
  {'_template': '9300cdc044d4b75151044340118ccf4efd976922',
    '_type': u'default',
    u'name': [u'Antarctica'],
    u'population': [u'0'],
    'url': 'http://example.webscraping.com/view/Antarctica-9'}
...
[example.webscraping.com] DEBUG: Retrying <GET
http://example.webscraping.com/edit/Antarctica-9> (failed 1 times): 500
Internal Server Error
```

这和“优化放置”小节中遇到的问题是一样的，因为 Portia 生成的项目使用了默认的 Scrapy 爬取设置，导致下载速度过快。我们仍然可以在设置文件中修改这些设置（文件位于 data/projects/new_project/spiders/settings.py）。不过，为了演示一些新方法，这次我们改为使用命令行进行设置。

```
(portia_example)$ portiacrawl portia/slyd/data/projects/new_project
example.webscraping.com -s CONCURRENT_REQUESTS_PER_DOMAIN=1 -s DOWNLOAD_DELAY=5
...
[example.webscraping.com] DEBUG: Crawled (200) <GET
```

```
http://example.webscraping.com/user/login?_next=%2Findex%2F1>
[example.webscraping.com] DEBUG: Crawled (200) <GET
http://example.webscraping.com/user/register?_next=%2Findex%2F1>
```

当运行这个放慢速度的爬虫时，就可以避免被封禁的问题了。不过，接下来同样也会遇到下载非必要网页（比如登录和注册页）这个降低效率的问题。默认情况下，Portia 生成的爬虫会爬取给定域名的所有 URL。要想只爬取特定 URL，可以配置右边栏面板中的 **Crawling** 选项卡，如图 8.6 所示。

图 8.6

这里，我们添加/index/和/view/作为爬虫跟踪模式，并且将/user/作为排除模式，这些都和之前 Scrapy 项目中的用法相似。如果勾选了底部的 **Overlay blocked links** 复选框，Portia 就会把跟踪链接高亮为绿色，排除链接高亮为红色，如图 8.7 所示。

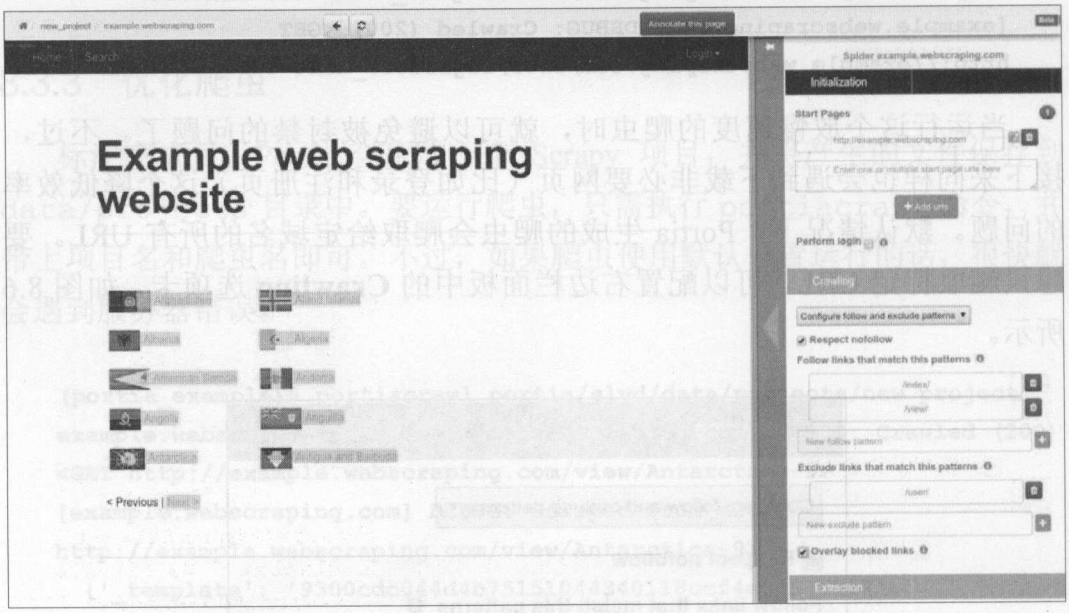


图 8.7

8.3.4　检查结果

现在就可以执行 Portia 生成的爬虫了，另外和之前一样，我们使用`--output`选项指定输出的 CSV 文件。

```
(portia_example)$ portiacrawl portia/slyd/data/projects/new_project
example.webscraping.com --output=countries.csv -s CONCURRENT_REQUESTS_
PER_ DOMAIN=1 -s DOWNLOAD_DELAY=5
```

当运行如上命令时，该爬虫将会产生和手工创建的 Scrapy 版本相同的输出。

Portia 是一个非常方便的与 Scrapy 配合的工具。对于简单的网站，使用 Portia 开发爬虫通常速度更快。相反，对于复杂的网站（比如依赖 JavaScript 的界面），则可以选择使用 Python 直接开发 Scrapy 爬虫。

8.4 使用 Scrapely 实现自动化抓取

为了抓取标注域，Portia 使用了 **Scrapely** 库，这是一款独立于 Portia 之外的非常有用的开源工具，该工具可以从 `https://github.com/scrapy/scrapely` 获取。Scrapely 使用训练数据建立从网页中抓取哪些内容的模型，并在以后抓取相同结构的其他网页时应用该模型。下面是该工具的运行示例。

```
(portia_example)$ python
>>> from scrapely import Scraper
>>> s = Scraper()
>>> train_url = 'http://example.webscraping.com/view/Afghanistan-1'
>>> s.train(train_url, {'name': 'Afghanistan', 'population': '29,121,286'})
>>> test_url = 'http://example.webscraping.com/view/United-Kingdom-239'
>>> s.scrape(test_url)
[{u'name': [u'United Kingdom'], u'population': [u'62,348,447']}]
```

首先，将我们想要从 `Afghanistan` 网页中抓取的数据传给 Scrapely，本例中是国家名称和人口数量。然后，在另一个国家页上应用该模型，可以看出 Scrapely 使用该模型返回了正确的国家名称和人口数量。

这一工作流允许我们无须知晓网页结构，只把所需内容抽取出来作为训练案例，就可以抓取网页。如果网页内容是静态的，在布局发生改变时，这种方法就会非常有用。例如一个新闻网站，已发表文章的文本一般不会发生变化，但是其布局可能会更新。这种情况下，Scrapely 可以使用相同的数据重新训练，针对新的网站结构生成模型。

在测试 Scrapely 时，此处使用的示例网页具有良好的结构，每个数据类型的标签和属性都是独立的，因此 Scrapely 可以正确地训练模型。但是，对于更加复杂的网页，Scrapely 可能在定位内容时失败，因此在其文档中会警告你应当“谨慎训练”。也许今后会有更加健壮的自动化爬虫库发布，不过现在仍需了解使用第 2 章中介绍的技术直接抓取网站的方法。

8.5　本章小结

本章首先介绍了网络爬虫框架 Scrapy，该框架拥有很多能够改善抓取网站效率的高级功能。然后介绍了 Portia，它提供了生成 Scrapy 爬虫的可视化界面。最后我们试用了 Scrapely，Portia 正是使用该库根据给定模型自动化抓取网页的。

下一章中，我们将应用前面学到的这些技巧来抓取现实世界中的网站。

第9章 总结

截止到目前，本书介绍的爬虫技术都应用于一个定制网站，这样可以帮助我们更加专注于学习特定技巧。而在本章中，我们将分析几个真实网站，来看看这些技巧是如何应用的。首先我们使用 Google 演示一个真实的搜索表单，然后是依赖 JavaScript 的网站 Facebook，接下来是典型的在线商店 Gap，最后是拥有地图接口的宝马官网。由于这些都是活跃的网站，因此读者在阅读本书时这些网站存在已经发生变更的风险。不过这样也好，因为这些例子的目的是为了向你展示如何应用前面所学的技术，而不是展示如何抓取指定网站。当你选择运行某个示例时，首先需要检查网站结构在示例编写后是否发生过改变，以及当前该网站的条款与条件是否禁止了爬虫。

9.1 Google 搜索引擎

根据第 4 章中 Alexa 的数据，google.com 是全世界最流行的网站之一，而且非常方便的是，该网站结构简单，易于抓取。

图 9.1 所示为 Google 搜索主页使用 Firebug 加载查看表单元素时的界面。

Google 国际化版本

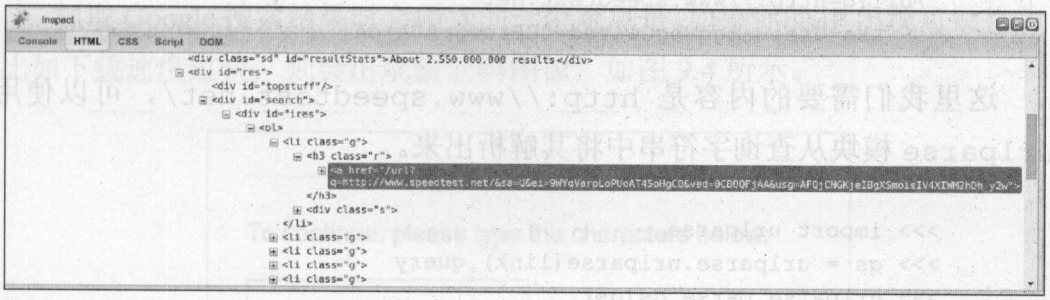
Google 可能会根据你的地理位置跳转到指定国家的版本。要想无论处于世界任何地方，都能使用一致的 Google 搜索，那么可以使用 Google 的国际化英文版本，其网址为 `http://www.google.com/ncr`。其中，ncr 表示**没有国家跳转（no country redirect）**。

图 9.1

可以看到，搜索查询存储在输入参数 `q` 当中，然后表单提交到 `action` 属性设定的`/search` 路径。我们可以通过将 test 作为搜索条件提交给表单对其进行测试，此时会跳转到类似 `https://www.google.com/search?q=test&oq=test&es_sm=93&ie=UTF-8` 的 URL 中。确切的 URL 取决于你的浏览器和地理位置。此外，还需要注意的是，如果开启了 Google 实时，那么搜索结果会使用 AJAX 执行动态加载，而不再需要提交表单。虽然 URL 中包含了很多参数，但是只有用于查询的参数 `q` 是必需的。当 URL 为 `https://www.google.com/search?q=test` 时，也能产生相同的结果，如图 9.2 所示。

图 9.2

搜索结果的结构可以使用 Firebug 来检查，如图 9.3 所示。


Inspect
Console
HTML
CSS
Script
DOM
About 2,550,000,000 results

<li class="g">
<h3 class="r">
<a href="/url?
q=http://www.speedtest.net/&sa=U&ei=9WYqVarpLoPUoAT45oHgCQ&ved=0CBQQFjAA&usg=AFQjCNGKjeIDgXSmoisIv4XIWN2hDn_y2w">
</h3>

<li class="g">
<li class="g">
<li class="g">

```
>>> results = tree.cssselect('h3.r a')
>>> results
[<Element a at 0x7f3d9affeaf8>,
  <Element a at 0x7f3d9affe890>,
  <Element a at 0x7f3d9affe8e8>,
  <Element a at 0x7f3d9affeaa0>,
  <Element a at 0x7f3d9b1a9e68>,
  <Element a at 0x7f3d9b1a9c58>,
  <Element a at 0x7f3d9b1a9ec0>,
  <Element a at 0x7f3d9b1a9f18>,
  <Element a at 0x7f3d9b1a9f70>,
  <Element a at 0x7f3d9b1a9fc8>]
```

到目前为止，我们已经下载得到了 Google 的搜索结果，并且使用 lxml 抽取出其中的链接。在图 9.3 中，我们发现链接中的真实网站 URL 之后还包含了一串附加参数，这些参数将用于跟踪点击。下面是第一个链接。

```
>>> link = results[0].get('href')
>>> link
'/url?q=http://www.speedtest.net/
    &sa=U&ei=nmgqVbgCw&ved=0CB&usg=ACA_cA'
```

这里我们需要的内容是 http://www.speedtest.net/，可以使用 urlparse 模块从查询字符串中将其解析出来。

```
>>> import urlparse
>>> qs = urlparse.urlparse(link).query
>>> urlparse.parse_qs(qs)
{'q': ['http://www.speedtest.net/'],
  'ei': ['nmgqVbgCw'],
  'sa': ['U'],
  'usg': ['ACA_cA'],
  'ved': ['0CB']}
>>> urlparse.parse_qs(qs).get('q', [])
['http://www.speedtest.net/']
```

该查询字符串解析方法可以用于抽取所有链接。

```
>>> links = []
>>> for result in results:
```

```
...     link = result.get('href')
...     qs = urlparse.urlparse(link).query
...     links.extend(urlparse.parse_qs(qs).get('q', []))
...
>>> links
 ['http://www.speedtest.net/',
  'https://www.test.com/',
  'http://www.tested.com/',
  'http://www.speakeasy.net/speedtest/',
  'http://www.humanmetrics.com/cgi-win/jtypes2.asp',
  'http://en.wikipedia.org/wiki/Test_cricket',
  'https://html5test.com/',
  'http://www.16personalities.com/free-personality-test',
  'https://www.google.com/webmasters/tools/mobile-friendly/',
  'http://speedtest.comcast.net/']
```

成功了！从 Google 搜索中得到的链接已经被成功抓取出来了。该示例的完整源码可以从 `https://bitbucket.org/wswp/code/src/tip/chapter09/google.py` 获取。

抓取 Google 搜索结果时会碰到的一个难点是，如果你的 IP 出现可疑行为，比如下载速度过快，则会出现验证码图像，如图 9.4 所示。

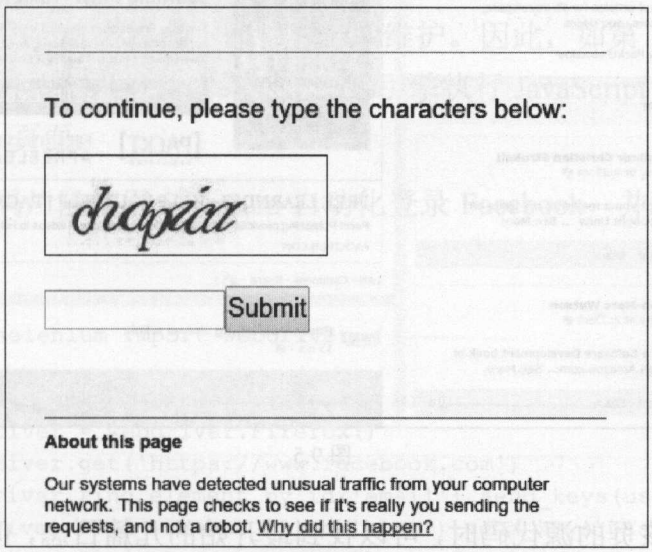


图 9.4

我们可以使用第 7 章中介绍的技术来解决验证码图像这一问题，不过更好的方法是降低下载速度，或者在必须高速下载时使用代理，以避免被 Google 怀疑。

9.2　Facebook

目前，从月活用户数维度来看，Facebook 是世界上最大的社交网络之一，因此其用户数据非常有价值。

9.2.1　网站

图 9.5 所示为 Packt 出版社的 Facebook 页面，其网址为 `https://www.facebook.com/PacktPub`。

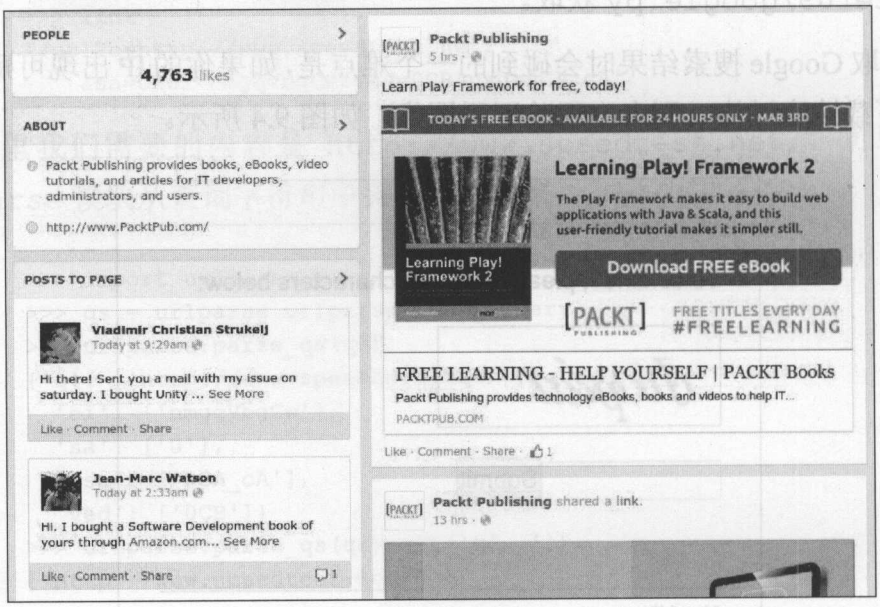


图 9.5

当你查看该页的源代码时，可以找到最开始的几篇日志，但是后面的日志只有在浏览器滚动时才会通过 AJAX 加载。另外，Facebook 还提供了一个移

动端界面，正如第 1 章所述，这种形式的界面通常更容易抓取。该页面在移动端的网址为 https://m.facebook.com/PacktPub，如图 9.6 所示。

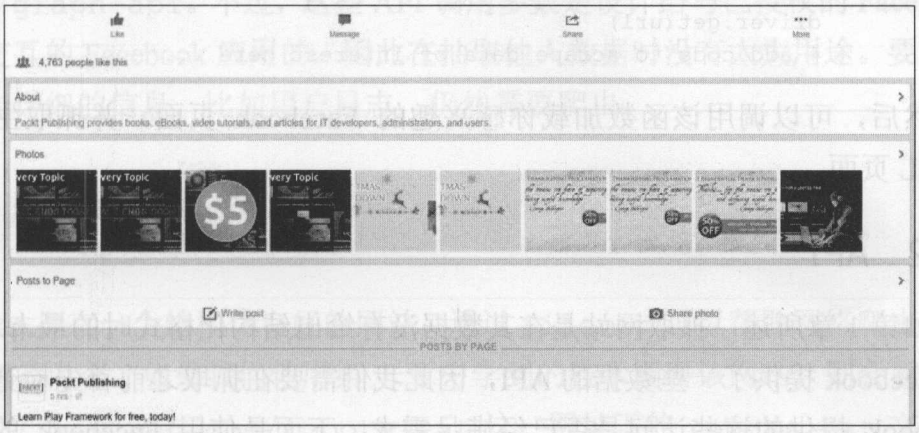


图 9.6

当我们与移动端网站进行交互，并使用 Firebug 查看时，会发现该界面使用了和之前相似的结构用于处理 AJAX 事件，因此该方法实际上无法简化抓取。虽然这些 AJAX 事件可以被逆向工程，但是不同类型的 Facebook 页面使用了不同的 AJAX 调用，而且依据我的过往经验，Facebook 经常会变更这些调用的结构，所以抓取这些页面需要持续维护。因此，如第 5 章所述，除非性能十分重要，否则最好使用浏览器渲染引擎执行 JavaScript 事件，然后访问生成的 HTML 页面。

下面的代码片段使用 Selenium 自动化登录 Facebook，并跳转到给定页面的 URL。

```
from selenium import webdriver

def facebook(username, password, url):
    driver = webdriver.Firefox()
    driver.get('https://www.facebook.com')
    driver.find_element_by_id('email').send_keys(username)
    driver.find_element_by_id('pass').send_keys(password)
    driver.find_element_by_id('login_form').submit()
    driver.implicitly_wait(30)
```

```
    # wait until the search box is available,
    # which means have successfully logged in
    search = driver.find_element_by_id('q')
    # now logged in so can go to the page of interest
    driver.get(url)
    # add code to scrape data of interest here ...
```

然后，可以调用该函数加载你感兴趣的 Facebook 页面，并抓取生成的 HTML 页面。

9.2.2　API

如第 1 章所述，抓取网站是在其数据没有给出结构化格式时的最末之选。而 Facebook 提供了一些数据的 API，因此我们需要在抓取之前首先检查一下 Facebook 提供的这些访问是否已经满足需求。下面是使用 Facebook 的图形 API 从 Packt 出版社页面中抽取数据的代码示例。

```
>>> import json, pprint
>>> html = D('http://graph.facebook.com/PacktPub')
>>> pprint.pprint(json.loads(html))
{u'about': u'Packt Publishing provides books, eBooks, video
    tutorials, and articles for IT developers, administrators, and
        users.',
u'category': u'Product/service',
u'founded': u'2004',
u'id': u'204603129458',
u'likes': 4817,
u'link': u'https://www.facebook.com/PacktPub',
u'mission': u'We help the world put software to work in new ways,
    through the delivery of effective learning and information
        services to IT professionals.',
u'name': u'Packt Publishing',
u'talking_about_count': 55,
u'username': u'PacktPub',
u'website': u'http://www.PacktPub.com'}
```

该 API 调用以 JSON 格式返回数据，我们可以使用 `json` 模块将其解析为 Python 的 `dict` 类型。然后，我们可以从中抽取一些有用的特征，比如公司名、详细信息以及网站等。

图形API还提供了很多访问用户数据的其他调用，其文档可以从Facebook的开发者页面中获取，网址为 https://developers.facebook.com/docs/graph-api。不过，这些API调用多数是设计给与已授权的Facebook用户交互的Facebook应用的，因此在抽取他人数据时没有太大用途。要想得到更加详细的信息，比如用户日志，仍然需要爬虫。

9.3 Gap

Gap拥有一个结构化良好的网站，通过Sitemap可以帮助网络爬虫定位其最新的内容。如果我们使用第 1 章中学到的技术调研该网站，则会发现在http://www.gap.com/robots.txt这一网址下的robots.txt文件中包含了网站地图的链接。

```
Sitemap: http://www.gap.com/products/sitemap_index.xml
```

下面是链接的Sitemap文件中的内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <sitemap>
        <loc>http://www.gap.com/products/sitemap_1.xml</loc>
        <lastmod>2015-03-03</lastmod>
    </sitemap>
    <sitemap>
        <loc>http://www.gap.com/products/sitemap_2.xml</loc>
        <lastmod>2015-03-03</lastmod>
    </sitemap>
</sitemapindex>
```

如上所示，Sitemap 链接中的内容仅仅是索引，其中又包含了其他Sitemap文件的链接。其他的这些Sitemap文件中则包含了数千种产品类目的链接，比如http://www.gap.com/products/blue-long-sleeve-shirts-for-men.jsp，如图 9.7 所示。

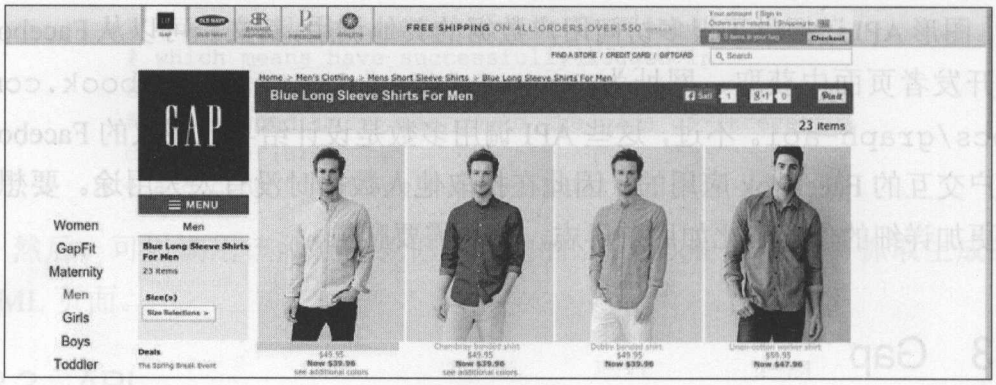


图 9.7

这里有大量要爬取的内容，因此我们将使用第 4 章中开发的多线程爬虫。你可能还记得该爬虫支持一个可选的回调参数，用于定义如何解析下载到的网页。下面是爬取 Gap 网站中 Sitemap 链接的回调函数。

```
from lxml import etree
from threaded_crawler import threaded_crawler

def scrape_callback(url, html):
    if url.endswith('.xml'):
        # Parse the sitemap XML file
        tree = etree.fromstring(html)
        links = [e[0].text for e in tree]
        return links
    else:
        # Add scraping code here
        pass
```

该回调函数首先检查下载到的 URL 的扩展名。如果扩展名为 .xml，则认为下载到的 URL 是 Sitemap 文件，然后使用 lxml 的 etree 模块解析 XML 文件并从中抽取链接。否则，认为这是一个类目 URL，不过本例中还没有实现抓取类目的功能。现在，我们可以在多线程爬虫中使用该回调函数来爬取 gap.com 了。

```
>>> from threaded_crawler import threaded_crawler
>>> sitemap = 'http://www.gap.com/products/sitemap_index.xml'
>>> threaded_crawler(sitemap, scrape_callback=scrape_callback)
Downloading: http://www.gap.com/products/sitemap_1.xml
```

```
Downloading: http://www.gap.com/products/sitemap_2.xml
Downloading: http://www.gap.com/products/
    cable-knit-beanie-P987537.jsp
Downloading: http://www.gap.com/products/
    2-in-1-stripe-tee-P987544.jsp
Downloading: http://www.gap.com/products/boyfriend-jeans-2.jsp
...
```

和预期一样，首先下载的是 Sitemap 文件，然后是服装类目。

9.4 宝马

宝马官方网站中有一个查询本地经销商的搜索工具，其网址为 https://www.bmw.de/de/home.html?entryType=dlo，界面如图 9.8 所示。

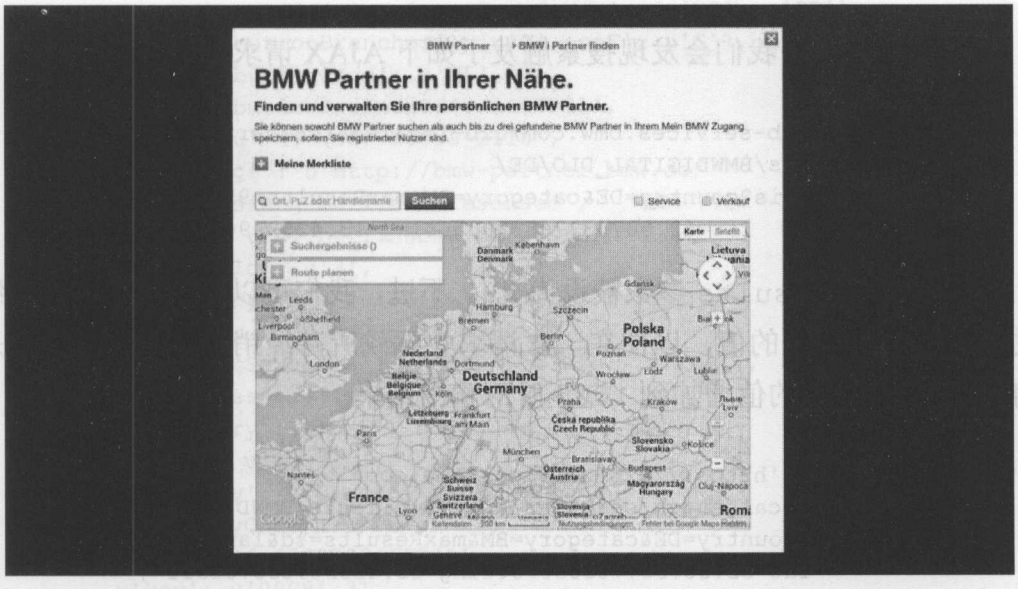


图 9.8

该工具将地理位置作为输入参数，然后在地图上显示附近的经销商地点，比如在图 9.9 中以 Berlin 作为搜索参数。

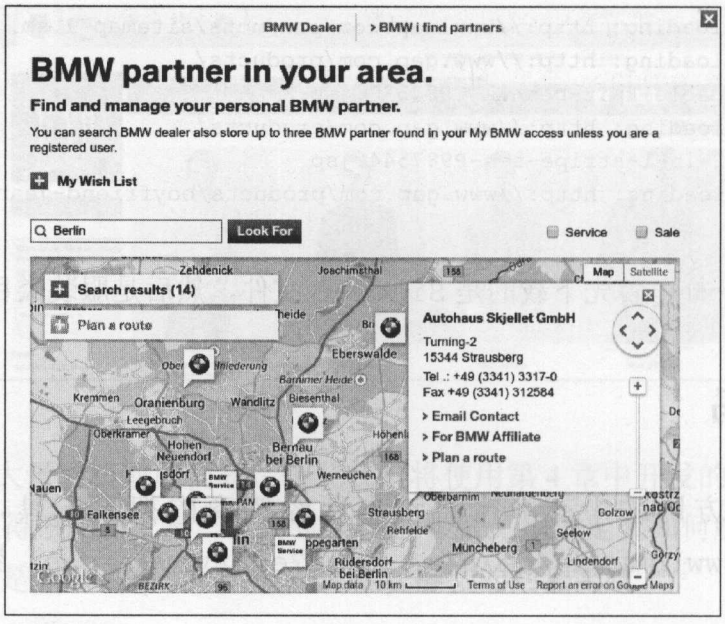


图 9.9

使用 Firebug，我们会发现搜索触发了如下 AJAX 请求。

```
https://c2b-services.bmw.com/c2b-localsearch/services/api/v3/
    clients/BMWDIGITAL_DLO/DE/
        pois?country=DE&category=BM&maxResults=99&language=en&
            lat=52.507537768880056&lng=13.425269635701511
```

这里，maxResults 参数被设为 99。不过，我们可以使用第 1 章中介绍的技术增大该参数的值，以便在一次请求中下载所有经销商的地点。下面是将 maxResults 的值增加到 1000 时的输出结果。

```
>>> url = 'https://c2b-services.bmw.com/
    c2b-localsearch/services/api/v3/clients/BMWDIGITAL_DLO/DE/
    pois?country=DE&category=BM&maxResults=%d&language=en&
        lat=52.507537768880056&lng=13.425269635701511'
>>> jsonp = D(url % 1000)
>>> jsonp
'callback({"status":{
...
})'
```

AJAX请求提供了**JSONP**格式的数据，其中JSONP是指**填充模式的JSON**（**JSON with padding**）。这里的填充通常是指要调用的函数，而函数的参数则为纯 JSON 数据，在本例中调用的是 `callback` 函数。要想使用 Python 的 `json` 模块解析该数据，首先需要将填充部分截取掉。

```
>>> import json
>>> pure_json = jsonp[jsonp.index('(') + 1 : jsonp.rindex(')')]
>>> dealers = json.loads(pure_json)
>>> dealers.keys()
 [u'status', u'count', u'translation', u'data', u'metadata']
>>> dealers['count']
731
```

现在，我们已经将德国所有的宝马经销商加载到JSON对象中，可以看出目前总共有731个经销商。下面是第一个经销商的数据。

```
>>> dealers['data']['pois'][0]
{u'attributes': {u'businessTypeCodes': [u'NO', u'PR'],
  u'distributionBranches': [u'T', u'F', u'G'],
  u'distributionCode': u'NL',
  u'distributionPartnerId': u'00081',
  u'fax': u'+49 (30) 20099-2110',
  u'homepage': u'http://bmw-partner.bmw.de/
  niederlassung-berlin-weissensee',
  u'mail': u'nl.berlin@bmw.de',
  u'outletId': u'3',
  u'outletTypes': [u'FU'],
  u'phone': u'+49 (30) 20099-0',
  u'requestServices': [u'RFO', u'RID', u'TDA'],
  u'services': []},
  u'category': u'BMW',
  u'city': u'Berlin',
  u'country': u'Germany',
  u'countryCode': u'DE',
  u'dist': 6.65291036632401,
  u'key': u'00081_3',
  u'lat': 52.562568863415,
  u'lng': 13.463589476607,
  u'name': u'BMW AG Niederlassung Berlin Filiale Wei\xdfensee',
  u'postalCode': u'13088',
  u'street': u'Gehringstr. 20'}
```

现在可以保存我们感兴趣的数据了。下面的代码片段将经销商的名称和经纬度写入一个电子表格当中。

```
with open('bmw.csv', 'w') as fp:
    writer = csv.writer(fp)
    writer.writerow(['Name', 'Latitude', 'Longitude'])
    for dealer in dealers['data']['pois']:
        name = dealer['name'].encode('utf-8')
        lat, lng = dealer['lat'], dealer['lng']
        writer.writerow([name, lat, lng])
```

运行该示例后，得到的 bmw.csv 表格中的内容类似如下所示。

```
Name,Latitude,Longitude
BMW AG Niederlassung Berlin Filiale
Weißensee,52.562568863415, 13.463589476607
Autohaus Graubaum GmbH,52.4528925,13.521265
Autohaus Reier GmbH & Co. KG,52.56473,13.32521
```

从宝马官网抓取数据的完整源代码可以从 https://bitbucket.org/wswp/code/src/tip/chapter09/bmw.py 获取。

翻译外文内容

你可能已经注意到宝马的第一个截图（见图 9.8）是德文的，而第二个截图（见图 9.9）是英文的。这是因为第二个截图中的文本使用了 Google 翻译的浏览器扩展进行了翻译。当尝试了解如何在外文网站中定位时，这是一个非常有用的技术。宝马官网在经过翻译后，仍然可以正常运行。不过还是要当心 Google 翻译可能会破坏一些网站的正常运行，比如依赖原始值的表单，其中的下拉菜单内容被翻译时就会出现问题。

在 Chrome 中，Google 翻译可以通过安装 Google Translate 扩展获得；在 Firefox 中，可以安装 Google Translator 插件；而在 IE 中，则可以安装 Google Toolbar。此外，还可以使用 http://translate.google.com 进行翻译，不过这样通常会打断原有功能，因为要从 Google 的网站中获取相关内容。

9.5 本章小结

本章分析了几个著名网站，并演示了如何在其中应用本书中介绍过的技术。我们在抓取 Google 结果页时使用了 CSS 选择器，在抓取 Facebook 页面时测试了浏览器渲染引擎和 API，在爬取 Gap 时使用了 `Sitemap`，在从地图中抓取所有宝马经销商时利用了 AJAX 调用。

现在，你可以运用本书中介绍的技术来抓取包含有你感兴趣数据的网站了。希望你能够和我一样享受这其中的力量！

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 0 使用积分 里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 S4XC5 使用优惠码，然后点击“使用优惠码”，即可在原折扣基础上享受全单9折优惠。（订单满39元即可使用，本优惠券只可使用一次）

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区	微信服务号	微信订阅号	官方微博	QQ 群：368449889

社区网址：www.epubit.com.cn

投稿 & 咨询：contact@epubit.com.cn

用Python写网络爬虫

作为一种便捷地收集网上信息并从中抽取出可用信息的方式，网络爬虫技术变得越来越有用。使用Python这样的简单编程语言，你仅需要使用少量编程技能就可以爬取复杂的网站。

本书作为使用Python来爬取网络数据的权威指南，讲解了从静态页面爬取数据的方法以及使用缓存来管理服务器负载的方法。此外，本书还介绍了如何使用AJAX URL和Firebug扩展来爬取数据，以及有关爬取技术的更多真相，比如使用浏览器渲染、管理cookie、通过提交表单从受验证码保护的复杂网站中抽取数据等。最后，本书使用Scrapy创建了一个高级网络爬虫，并对一些真实的网站进行了爬取。

本书的目标读者

本书是为打算构建可靠的数据爬取解决方案的开发人员写作的，本书假定读者具有一定的Python编程经验。当然，具备其他编程语言开发经验的读者也可以阅读本书，并理解书中涉及的概念和原理。

本书介绍了如下内容：

- 通过跟踪链接来爬取网站；
- 使用lxml从页面中抽取数据；
- 构建线程爬虫来并行爬取页面；
- 将下载的内容进行缓存，以降低带宽消耗；
- 解析依赖于JavaScript的网站；
- 与表单和会话进行交互；
- 解决受保护页面的验证码问题；
- 对AJAX调用进行逆向工程；
- 使用Scrapy创建高级爬虫。

美术编辑：董志桢

分类建议：计算机／程序设计／Python
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43179-0

ISBN 978-7-115-43179-0
定价：45.00 元